**Partie II : Un outil indispensable : les tables de hachage.**

a) Définitions.

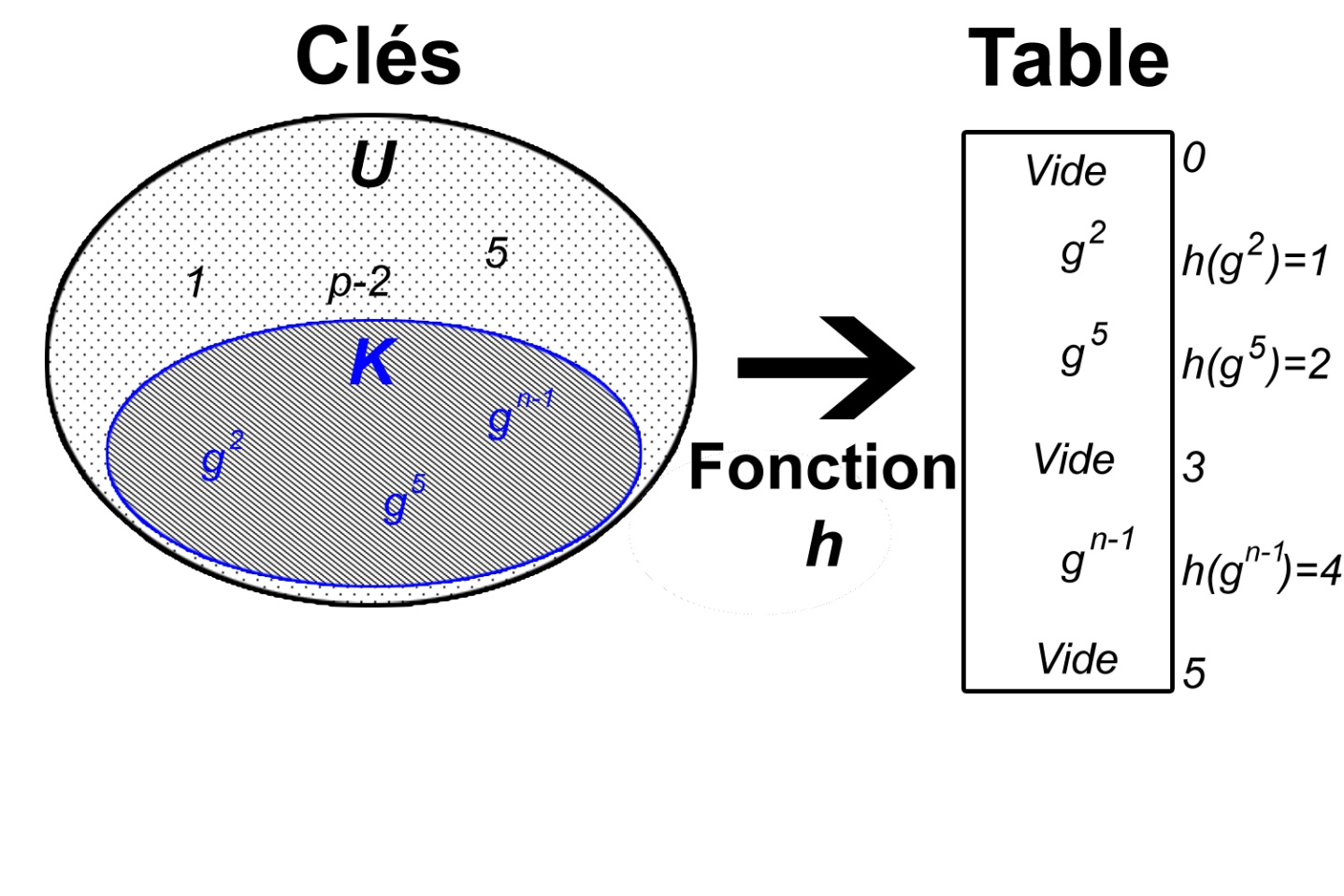
Le but des tables de hachage va être d’assurer une complexité d’insertion et de recherche qui soit **en moyenne** constante, c'est-à-dire qu’elle ne dépende pas de *n* (donc de *p*). Les complexités traitées ici seront donc des complexités en moyenne et pas dans le pire des cas (dans le pire des cas on peut pas faire mieux que pour la recherche).

On définit « l’univers des clés» et on note l’ensemble des clés de *U* qu’on doit stocker.

*Définition 2 :* On définit une « table de hachage » comme un tableau de taille dont la case numéro i (nommée « alvéole ») est :

* Soit vide.
* Soit contient un élément tel que .

L’application est appelée « fonction de hachage ».

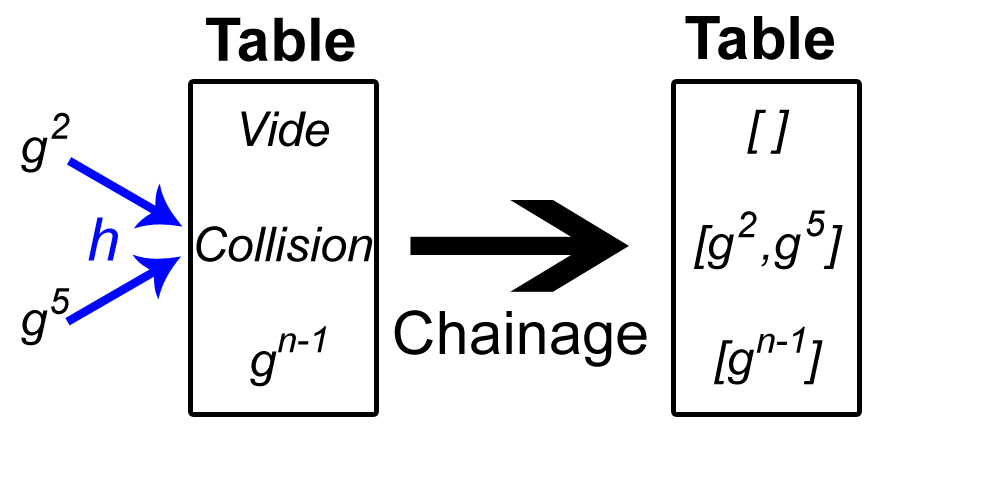


*b) Gestion des collisions par chainage.*

Cependant, on se rend facilement compte qu’un tel fonctionnement présente un problème lorsque deux clés possèdent la même image par la fonction *h.*

*Définition 3:* Soit . On dit qu’il y a « collision » si et .

Afin de gérer les collisions, on va « chainer » les clés qui possèdent la même image par *h*. Pour cela, on place dans la table de hachage des listes qui peuvent soit être vides soit contenir un ou plus d’éléments.



Les deux opérations qui nous intéressent, l’insertion et la recherche d’un élément, s’effectuent alors la façon suivante :

*Insertion*: Lorsqu’on insère un élément , on le place en tête de la liste qui occupe la position dans la table de hachage. Le cout d’une telle opération est le même avec ou sans chainage.

*Recherche :* Lorsqu’on recherche un élément , on le recherche dans la liste qui occupe la position dans la table de hachage. Le cout d’une recherche avec chainage est théoriquement plus important que la simple observation de la case lorsqu’il n’y a pas de chainage. Cependant, si on repartit bien les différentes clés à l’aide d’une « bonne » fonction de hachage les listes occupant les cases de la table de hachage ont une taille très inferieure au nombre de clés stockées ce qui justifie l’efficacité de cette méthode par rapport à la simple utilisation d’une liste.

*c) Etude d’une fonction de hachage.*

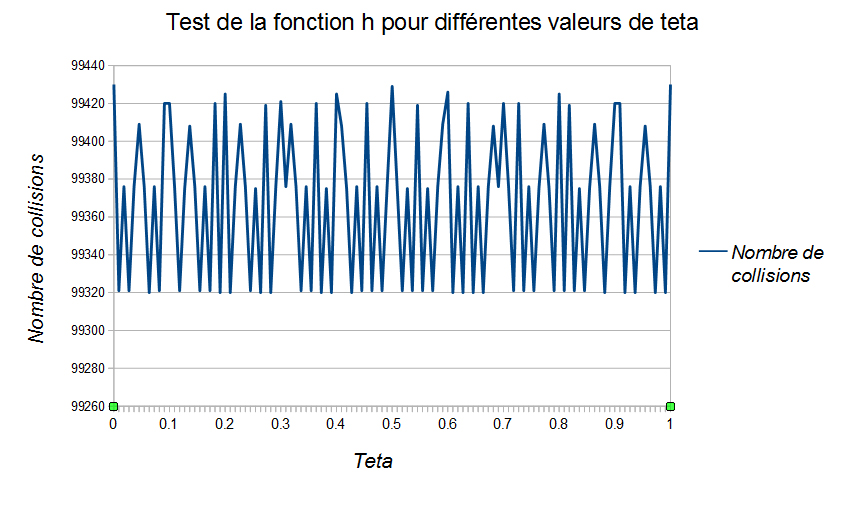
Une « bonne » fonction de hachage non seulement doit permettre de calculer des clés en un temps relativement court, elle doit aussi être le plus uniforme possible afin de prévenir un maximum les collisions qui ralentissent le processus de recherche.

Un des types de fonctions de hachage utilisé de nos jours est de la forme suivante :

avec une constante.

Le terme renvoi les décimales de la valeur . On multiplie après par la taille du tableau puis on prend la partie entière, ce qui donne bien un indice entre 0 et .

Afin de déterminer la constante la mieux adaptée (celle qui rend *h* le plus uniforme possible), j’ai simulé informatiquement le stockage de *N* variables entre 0 et *N-1* générées aléatoirement pour différentes valeurs de dans un tableau de taille *N* tout en comptant le nombre de collisions. Pour *N=99431*, j’ai obtenu le graphique suivant :



Malheureusement, cette étude ne m’a pas permis d’apporter un résultat concluant pour plusieurs raisons :

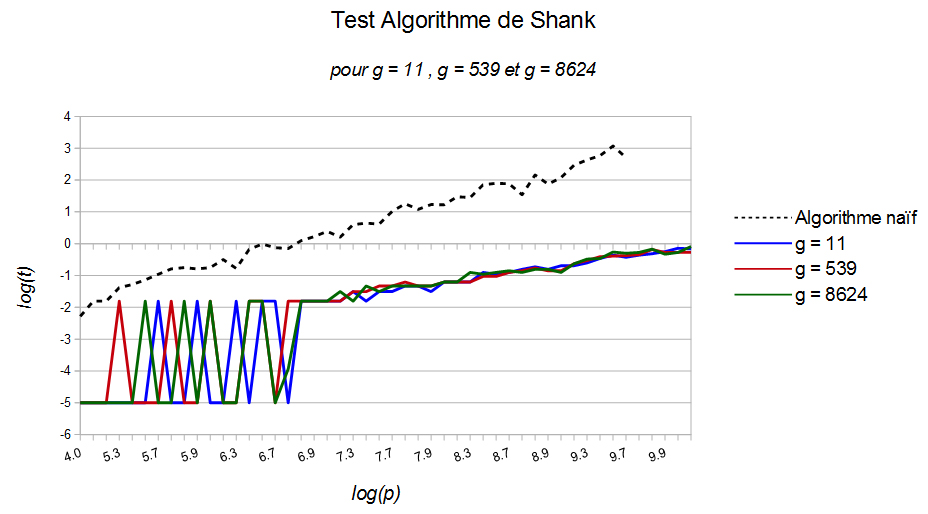
- Tout d’abord, les résultats présents dans le graphique sont particulièrement mauvais : on a au minimum *99 320* collisions.

- Deuxièmement, l’efficacité de la fonction *h* varie grandement entre deux valeurs très proches de ce qui m’a empêché de distinguer une plage de valeurs sur laquelle on pourrait trouver une valeur plus optimale.

Cependant, des études menées par Donald Knuth montrent que une valeur optimale de est la suivante : . J’ai soumis la fonction *h*  au même test pour cette valeur et j ai que obtenu 41337 collisions (soit la moitié par rapport à avant) !

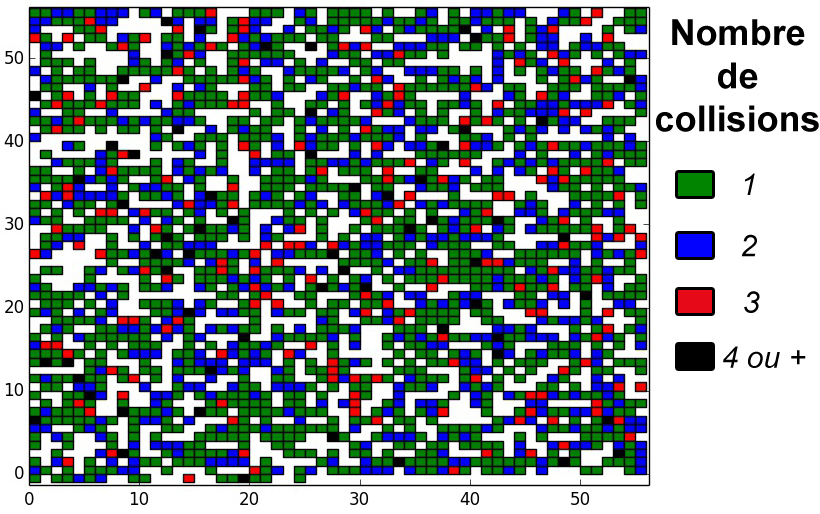
**Partie III : efficacité et limites de l’algorithme de Shank.**

En implémentant l’algorithme de Shank à l’aide de tables de hachage et en répartissant les clés avec la fonction *h\_opt* on obtient des résultats largement plus efficaces que l’algorithme naïf comme le montre l’étude de l’algorithme sur les mêmes valeurs que celles utilisées pour l’algorithme naïf :



Là où le temps d’exécution de l’algorithme naïf est de secondes, l’algorithme de Shank ne met même pas une seule seconde pour renvoyer un résultat.

J’ai même pu calculer le logarithme discret pour des valeurs de p de l’ordre de sans problème.



Cependant, lorsque *p* devient trop grand, on ne peut pas générer autant de cases que d’éléments à stocker car l’occupation mémoire serait trop importante. Il faut alors générer moins de cases pour le même nombre d’éléments.