Introduction aux techniques de hachage

Une revue des différentes techniques de hachage axées sur les données et la sécurité.

Gabin Chognot

UQAC
UNIVERSITÉ DU QUÉBEC
À CHICOUTIMI

2024-12-07

Contents

1 Introduction

1.1 Avant-propos

Ce texte est un résumé et une traduction de l'article de recherche "*Hashing Techniques: A Survey and Taxonomy*" publié en 2017 par *LIANHUA CHI* et *XINGQUAN ZHU*. Il vise à présenter les différentes techniques de hachage abordées dans l'article original, de manière plus accessible et plus digeste, en s'épargnant une large partie des subtilités mathématiques des concepts. Les méthodes mentionnées le sont de manière simplifiée mais permettant toujours de comprendre leur cadre général de fonctionnement. De plus, quand des ressources supplémentaires ont été utilisées à la traduction et au résumé de cet article, ou quand elles ont été jugées nécessaires à sa compréhension, celles-ci ont été ajoutées en hyperlien. Ces ressources sont en français quand elles existent, en anglais sinon.

S'agissant d'une traduction, le choix a été fait de fournir les termes en français quand ceux-ci sont suffisamment usités, ou d'en créer une traduction au besoin, en fournissant le terme original entre parenthèses, ainsi que son acronyme quand l'usage de celui-ci tend à remplacer le terme complet. Ce texte reste malgré tout, par la complexité du sujet et la nécessité des prérequis de base à propos du hachage et des notions de complexité, un texte destiné à ceux ayant déjà les notions essentielles du domaine des structures de données.

1.2 Introduction au hachage

Le hachage est un concept né en 1953, 3 ans après le premier ordinateur, dont le mot inspiré par son premier sens de découpage. En effet, le hachage, dont l'importance très rapidement reconnue en a fait un élément central en informatique, vise à relier des données d'entrée à des données de sortie dans un espace de dimension inférieure, en coupant, ou hachant, ces données d'entrée. On utilise aujourd'hui le hachage dans un vaste ensemble d'applications, du traitement d'images à la signature de messages, en passant par la recherche d'adresses IP ou le calcul de sommes de contrôles pour assurer l'intégrité d'une information.

Ainsi, on peut définir une fonction de hachage au travers de ses valeurs de hachages, c'est à dire : Soit $m \in \mathbb{N}$ tel que $[\![0; m]\!]$ soit un intervalle fixe, soit $n \in \mathbb{N}$ tel que $X = \{x_k; k \in [\![O; n]\!]\} \in \mathbb{R}^D$ soit un ensemble de données, et soit $h : \mathbb{R} \to \mathbb{N}$ une fonction de hachage, alors $h(X) = [h(x_1), h(x_2), ..., h(x_n)] \in [0; m]$ sont appelés les valeurs hachées de X.

Également, bien que ces concepts soient peu repris dans ce document à visée généraliste, il est utile de définir également les concepts de voisins, pour donner du contexte à vos futures recherches :

- Plus proche voisin (PPV) : Soit $n \in \mathbb{N}$ tel que $X = \{x_k; k \in [\![O; n]\!]\} \in \mathbb{R}^D$ soit un ensemble de données, et $q \in \mathbb{N}$ tel que $x_q \in X$, alors PPV est le point (ou l'ensemble des points) de X le(s) plus proche(s) de q.
- Plus proche voisin approximatif (PPVA): Soit $n \in \mathbb{N}$ tel que $X = \{x_k; k \in [\![O; n]\!]\} \in \mathbb{R}^D$ soit un ensemble de données, et $(q, a) \in \mathbb{N}^2$ tels que $(x_q, x_a) \in X^2$. Alors $PPVA$ est le x_a tel que pour une distance ϵ et pour tout $x \in X$, $d(x_a, x) \leq (1 + \epsilon)d(x_q, x)$.

On reconnaît deux usages du hachage :

1. Extraire ou comparer des données d'une grande base, où l'efficience d'accès très importante, c'est le hachage axé sur les données. On distingue deux types de hachage axé sur les données : le hachage indépendant des données, où la fonction de hachage se déploie sans entraînement préalable sur les données à hacher, et le hachage dépendant des données, où la fonction de hachage demande un entraînement sur des données entièrement, partiellement, ou pas labellisées.
2. Vérifier l'intégrité des données, où la sécurité est très importante, c'est le hachage axé sur la sécurité. On distingue deux types de hachage axé sur la sécurité : le hachage cryptographique, où les fonctions de hachage sont cryptographiquement sûres et surtout utilisées dans un contexte de signature, et le hachage non-cryptographique, où les fonctions de hachage ne sont pas cryptographiquement sûres, et sont utilisées pour des opérations de vérification d'intégrité de données.

Comme vous le verrez dans le reste du texte, on utilise parfois du hachage axé sur les données dans un contexte de sécurité informatique, et vice-versa. La classification présentée précédemment concerne uniquement l'objectif mathématique attribuée à la fonction de hachage, et non son utilisation pratique.

2 Hachage axé sur les données

De manière générale, ce type de hachage est principalement utilisé pour du traitement d'images, considérant à la fois la popularité récente de ce domaine, et la grande dimensionnalité et et les grands volumes de données inhérents à celui-ci. Plus particulièrement, les implémentations se concentrent généralement sur la récupération d'images. Une majorité des méthodes présentées ci-dessous servent partiellement ou entièrement à ce type d'opérations.

On retrouve également des opérations de hachage axées sur les données dans les réseaux, pour recherche d'adresses IP et de routes pour améliorer les performances, la classification et la surveillance des paquets réseaux dans le cadre de détection d'intrusions… Également dans la lecture, la recherche, ou le partitionnement de graphes, autant dans les réseaux sociaux qu'en chimie, en biologie, ou en informatique de la santé. Enfin, on en retrouve dans le traitement de texte, ou encore multimédia (comparaison d'empreintes digitales)…

Pour la suite, on distingue deux types de hachage axé sur les données : le hachage indépendant des données, et le hachage dépendant des données.

2.1 Hachage indépendant des données

Cette technique utilise une fonction prédéfinie, qui n'a pas besoin de s'entraîner sur les données à hacher pour fonctionner. Malgré tout, certaines apprennent la distribution des données pour s'améliorer, comme le hachage sensible à la localisation (*locality-sensitive hashing*), ou l'apprentissage pour le hachage (*learning for hashing*). On distingue quatre types de hachage indépendant des données :

1. La Projection aléatoire (*Random Projection*).
2. Le Hachage sensible à la localisation (*Locality-Sensitive Hashing*).
3. L'Apprentissage pour le hachage (*Learning for Hashing*).
4. La Projection structurée (*Structured Projection*).

2.1.1 Projection aléatoire

On définit une fonction de hachage aléatoire comme tel : soit U un ensemble de dimension d (les données), V un ensemble de dimension k (les hachages), avec $d \gg k$, et $h : U \to V$ une fonction de hachage tel que pour un x donné, $h(x)$ soit une valeur aléatoire. Rapidement, on peut remarquer qu'une fonction de ce type demande un espace de stockage de $d \times \log k$ bits, ce qui peut rapidement devenir très grand. On introduit donc également les fonctions fixées, où l'idée est de définir un espace restreint dans l'espace des fonctions aléatoires, qui contient uniquement des fonctions aléatoires à longueur de sortie fixe, dont on arrive donc à calculer le résultat en temps constant.

Parmi les implémentations on peut citer le Hachage à Projection Aléatoire (*Random Projection Hashing*), ou le Hachage Universel (*Universal Hashing*), qui sont tous les deux en $O(n)$, le premier étant également à espace constant, et le deuxième à espace linéaire.

2.1.2 Hachage sensible à la localisation

Si les fonctions de hachage aléatoire sont simples et efficaces, elles sont aussi instables. Deux données proches peuvent avoir des hachages très différents, et on a parfois besoin, pour atteindre de bonnes performances, de s'intéresser aux données hachées. Pour cela, on définit le *LSH* (*Locality-Sensitive Hashing*) comme tel : soit $n \in \mathbb{N}$ tel que $X = \{x_k; k \in [\![O; n]\!]\} \in \mathbb{R}^D$ soit un ensemble de données, et $H : \mathbb{R}^{D \times K} \to \{0; 1\}^K$ un ensemble de K fonctions de hachage, qui pour un $x \in X$ donné, renvoie un vecteur de K bits.

LSH préserve la localité, c'est à dire que $P\{H(x) = H(y)\} = \text{sim}(x, y)$, où sim est la proximité, une distance euclidienne par exemple. *LSH* permet aussi de prédire la probabilité de collision, c'est à dire la probabilité que $H(x) = H(y)$ pour deux données x et y données, qui est relativement élevée par rapport à une fonction de hachage aléatoire, en partie à cause de la localité. Enfin, *LSH* s'exécute en temps linéaire, donc en $O(n)$, et utilise un espace mémoire linéaire.

Mais, malgré ses avantages, *LSH* a un inconvénient majeur, l'inefficacité de son hachage, qui relève de :

1. L'impossibilité, à cause de la génération aléatoire des fonctions h, de d'assurer l'efficacité.
2. La longueur des codes dans les tables de hachage, qui permettent de garantir la proximité (souvent la concaténation des résultats de plusieurs fonctions), au détriment de la vitesse de recherche.

Parmi les implémentations basées sur *LSH*, on peut citer :

- *MinHash*, un algorithme en $O(n)$ et en espace linéaire, qui permet la comparaison rapide entre deux ensembles (par exemple deux textes) en calculant pour chacun un ensemble de hachages avec diverses fonctions (permutations), et en comparant les deux valeurs minimales obtenues. Pour un nombre de permutations suffisamment élevé, un *MinHash* équivaut à l'indice de Jaccard [1], couramment utilisé en statistiques. Le *MinHash* pondéré (*Weighted MinHash*) permet, en attribuant un poids à chaque élément, de prendre en compte son importance ou sa fréquence.
- Hachage basé sur un noyau invariant par décalage (*Shift-Invariant Kernel-Based Hashing*), étend la localité de LSH en calculant la distance avec un noyau invariant (c'est à dire une fonction de proximité insensible aux translations ou décalage), en l'occurrence grâce à la distance de Hamming [2].
- Hachage par sous-arbres imbriqués (*Nested Subtree Hashing*, *NSH*), une version de LSH appliqué aux graphes et aux arbres, où la localité est déterminée par la proximité des nœuds et des sous-arbres, pour permettre de chercher facilement dans ces structures de données.
- Hachage par discrimination de clique (*Discriminative Clique Hashing*), est une alternative à *NSH*, qui utilise la proximité des cliques (sous-graphes denses, aux nœuds très connectés) pour déterminer la localité. Il est par exemple utile dans l'analyse de graphes sociaux, pour identifier des communautés.

2.1.3 Apprentissage pour le hachage

L'Apprentissage pour le Hachage (*Learning For Hashing*, *LFH*) regroupe un ensemble de méthodes qui apprennent des données à hacher, pour préserver certaines propriétés comme la proximité. Cette famille se base sur des espaces latents [3], un type d'ensemble mathématique, de dimension réduite par rapport à un ensemble initial, à l'intérieur duquel les données similaires sont positionnées proches les unes des autres. On peut par exemple s'en servir pour représenter un graphe en une dimension.

Les différentes implémentations, détaillées ci-dessous, visent toutes à apprendre un nouvel espace latent pour les données, pour que la représentation des données hachées révèle ou conserve la similarité de celles-ci :

- *Fastmap* projette en une dimension des données en deux dimensions. Pour un espace X, on choisit deux points x_1 et x_2 arbitraires définis comme deux points de pivots, et on projette les $x \in X$ sur la droite passant par ces deux points. L'espace des projections est donc un espace euclidien [4] de dimension 1.
- *MetricMap* étend le principe de *Fastmap* en utilisant plusieurs paires de points de pivots, et projette donc sur un espace pseudo-euclidien [5]. En conséquence, *MetricMap* est très intéressant pour les espaces X non-euclidiens [6].
- *BoostMap* améliore *FastMap* et *MetricMap* en optimisant les opérations de réduction, et en préservant mieux que ses prédécesseurs la similarité après projection. *BoostMap* n'a pas besoin de mesure de distance et se repose sur les plongements [7]. Un plongement, ou morphisme injectif (dans le bon contexte, le mot a plusieurs sens), désigne un sous-objet d'un autre objet. *BoostMap* identifie un ensemble P de plongements en une dimension dans un espace X, en utilisant une paire de points de pivots ou un objet de référence. Les plongements, appelés classifieurs faibles, vont être combinés en un espace à $|P|$ dimensions, appelé classifieur fort, grâce à l'algorithme AdaBoost [8][9], permettant de combiner leurs informations respectives pour améliorer la précision de la classification. *BoostMap* fait ainsi appel à des outils classiques de *machine learning*.

2.1.4 Projection structurée

L'ensemble des méthodes présentées jusqu'à présent fonctionnent très bien pour le hachage d'espaces de faibles dimensions, mais leurs performances se dégradent quand elles sont appliquées à des espaces de grande dimension. Pour ces dernières, on utilise des structures qui partitionnent l'espace haché, l'indexent de manière multi-dimensionnelle.

La projection structurée est un ensemble de méthode de partitionnement de l'espace haché grâce à des règles pré-définies, contrairement à certaines méthodes de partitionnement qui s'adaptent aux données présentées.

On peut citer parmi les découpages possibles :

- *Quadtree* [10][11], un type d'arbre où chaque nœud a exactement un fils

- Courbe de Hilbert [12] (*Hilbert curve*), un type de courbe de remplissage [13] remplissant un carré. Cette courbe, pour donner un exemple compréhensible en deux dimensions, parcourt un tableau de données en S (à la manière d'un déplacement dans le célèbre jeu *Snake*), de telle sorte à ce que deux éléments proches dans le tableau soient aussi proches que possible sur la courbe.
- Courbe en Z [14][15] (*Z-curve*), un autre type de courbe de remplissage qui vise à répondre à un problème majeur des courbes de Hilbert : A mesure que la dimension de l'espace partitionné augmente, les subdivisions de celui-ci en sous-espaces de faible dimensions augmentent également, et avec elles la complexité du calcul des clefs, c'est à dire les indices qui lient la position d'un élément dans l'espace original et dans la courbe de Hilbert. La courbe en Z répond à ce problème en parcourant le tableau en Z et en le partitionnant en une dimension. Par exemple, toujours en deux dimensions, au lieu de "faire des virages" comme la courbe de Hilbert, elle parcourt toute une ligne, puis revient au début de la ligne d'après, et recommence, dessinant ainsi une suite de Z.

2.2 Hachage dépendant des données

Le hachage dépendant des données regroupe l'ensemble des familles de fonctions de hachage définies de manière unique pour un ensemble de donnés visées, elles s'entraînent sur celles-ci. Ce type de hachage est très sensible aux données hachées, mais fourni un temps de calcul très rapide, avec très peu de consommation de ressources.

On peut distinguer trois classes de familles, selon la labellisation des données d'entraînement :

1. Hachage non-supervisé (*Unsupervised hashing*) : Pas de labels.
2. Hachage semi-supervisé (*Semi-supervised hashing*) : Des labels partiels, qui indique par exemple simplement la proximité de deux données.
3. Hachage supervisé (*Supervised hashing*) : Chaque donnée est labellisée.

La majorité de ces méthodes de hachage sont en complexité spatiale de type espace de Hamming. On entend ici par espace de Hamming un espace composé de l'ensemble des 2^N mots binaires de longueur N. Étant dépendantes des données à hacher, calculer leur complexité temporelle n'est ni trivial ni généralisable à l'ensemble de ces méthodes, et l'information doit donc est donnée méthode par méthode.

2.2.1 Hachage non-supervisé

Les méthodes de hachage non-supervisées ne se basent sur absolument aucun label, pas même partiel. Elles génèrent généralement des codes binaires à partir des données à hacher, permettant de préserver les informations de proximité en construisant des codes proches pour des données proches. On distingue deux types de hachage non-supervisé selon le type de fonction utilisé :

1. Le Hachage spectral (*Spectral hashing*)
2. Le Hachage linéaire non-supervisé (*Unsupervised linear hashing*)

2.2.1.1 Hachage spectral

L'une des méthodes les plus populaires. Grâce à une fonction de hachage, elle compresse chaque donnée en un code binaire. Par exemple, à partir d'une liste de données, on obtient la même liste, composée de codes binaires. Elle réorganise ensuite ces codes binaires de manière à chaîner les données au même code, et à trier la table de hachage. Les codes sont ensuite remplacés par leur valeur de hachage, et la table de hachage est triée. L'explication donnée ici est généralisable aux dimensions supérieures à 1.

Plus précisément, pour réaliser cette compression, il faut d'abord réaliser une analyse en composantes principales [16] sur les données, pour réduire leur dimensionnalité. Ce processus est très gourmand en ressource, compliquant l'entraînement de l'algorithme. De plus, le hachage est lui aussi complexe, donc lent, souvent trop par rapport à la rapidité à l'usage demandée par les applications pratiques de ce type de hachage, puisque le hachage spectrale se fait en temps polynomial, en $O(n^x)$. Pour combler ces lacunes, le hachage spectral utilise des codes binaires courts pour accélérer la compression, mais au détriment de la précision de la compression.

Pour résoudre les problèmes évoqués, plusieurs méthodes existents, qui étendent le hachage spectral, comme le Hachage Spectral Multidimensionnel (*Multidimensional Spectral Hashing*), qui utilise une fonction de hachage multidimensionnelle, plus stable en performances que le hachage spectral classique. D'autres versions calculent la compression en comparant les données par paire, ou en calculant des produits scalaires…

2.2.1.2 Hachage linéaire non-supervisé

Le Hachage Linéaire Non-Supervisé, au nom transparent, englobe un ensemble de méthodes de hachage aux fonctions linéaires, obtenues à partir de l'analyse des données de l'espace de hachage. La plupart de ces fonctions utilisent les propriétés spectrales [17] (les valeurs propres) des matrices d'affinité [18] (qui capturent la proximité des données) pour générer des codes binaires, plus simples à traiter et à stocker.

Parmi les implémentations, on peut citer le Hachage basé sur Graphe d'Ancrage (*Anchor Graph Hashing*, *AGH*), en $O(n)$, qui analyse la structure de voisinage dans un graphe de données pour déterminer la fonction de hachage linéaire adaptée. Plus précisément, *AGH* choisit des points d'ancrage dans le nuage de données, en identifiant des *clusters* de données et leur point central respectif, qui permettent d'obtenir les vecteurs propres du graphe. Les informations de proximité données par ces vecteurs propres à propos des points d'ancrage est ensuite extrapolée à l'ensemble des données, permettant un temps d'entraînement linéaire et un temps de hachage constant.

Un moyen simple d'améliorer les performances de ces algorithmes est d'augmenter la taille des codes binaires, pour augmenter la précision de la compression, ce qui augmente d'autant la complexité spatiale. D'autres solutions ont été proposées à ce problème, comme le Hachage Parfait Minimal (*Minimal Perfect Hashing*, *MPH*) qui utilise des filtres de Bloom pour diminuer le nombre d'accès mémoire à $O(1)$, *HashMem* [19] qui hache les données en mémoire pour accélérer le processus, ou le Hachage Compressé [20] (*Compressed Hashing*) qui utilise un mélange de codage parcimonieux [21][22] et de d'acquisition comprimée [23][24].

Enfin, pour aller plus loin, parmi les autres méthodes, on pourrait citer la Quantification des produits (*Product Quantization*), qui divise les données en sous-ensembles, et le Codage binaire basé sur la quantification angulaire *(Angular Quantization-Based Binary Coding, AQBC)*, adaptés aux ensembles de grande dimension. Ou encore le Hachage Sphérique (*Spherical Hashing*), le Hachage Isotrope (*Isotropic Hashing*), le Hachage de Manhattan (*Manhattan Hashing*) qui utilise la distance du même nom, le Hachage à Double vue Prévisible (*Predictable Dual-View Hashing*), le Hachage à Collecteur Inductif (*Inductive Collector Hashing*), le Hachage Localement Linéaire (*Locally Linear Hashing*, *LLH*) en $O(n)$, le Hachage de Voisinage (*Neighborhood Hashing*), le Hachage préservant la Topologie (*Topology-Preserving Hashing*)…

2.2.1.3 Hachage non-linéaire non-supervisé

Le hachage non-linéaire non-supervisé utilise des fonctions non-linéaires telles que les noyaux [25]. Parmi les méthodes de cette famille, le *LSH* Kernelisé (*Kernelized LSH*), étend le principe du *LSH* vu ci-dessus, en construisant un hyperplan aléatoire, et en, grâce à une fonction noyau, distribuant les données dans un espace à haute dimension. Le *KLSH* fait ensuite appel au théorème central limite pour approximer la distribution en une gaussienne, de laquelle il tire un sous-ensemble d'éléments appelés plus-proches-voisins (définis en introduction). A partir desquels il applique ensuite l'opération de hachage sensible à la localisation. *KSLH* tourne en temps linéaire, c'est à dire en $O(n)$.

L'un des avantages de *KLSH* est l'absence de suppositions sur la structure initiale des données, ce qui le rend par exemple adapté à la recherche d'images. Des dérivés ont été proposés pour répondre à la faiblesse de *KLSH*, respectivement dans le cadre de codes binaires courts, et à la tendance des techniques de hachage à n'utiliser qu'une caractéristique des données. D'une part une variante de *KLSH* de 2010, et d'autre part le Hachage Kernélisé à Caractéristiques Multiples (*Multiple-Feature Kernelized Hashing*).

2.2.2 Hachage semi-supervisé

Dans le hachage semi-supervisé, on utilise à la fois des données labellisées, et des données non-labellisées, pour entraîner le modèle de hachage. Parmi les méthodes connues, nous verrons :

- Les méthodes de hachage semi-supervisé linéaires :
  - Le Hachage Semi-Supervisé (*Semisupervised hashing*, *SSH*).
  - Le Hachage Discriminant Semi-Supervisé (*Semisupervised Discriminant Hashing*, *SSDH*).
  - Le Hachage Semi-Supervisé Préservant la Topologie (*Semisupervised Topology-Preserving Hashing*, *STPH*).
- Les méthodes de hachage semi-supervisé non-linéaire :
  - La Partition à Marge Maximale Régularisée par Label (*Label-Regularized Maximum Margin Partition*, *LAMP*).
  - L'Apprentissage par Projection Séquentielle *Bootstrap* pour le Hachage Non-Linéaire Semi-Supervisé (*Sequential Bootstrapped Projection Learning for Semi-Supervised Nonlinear Hashing*, *Bootstrap-NSPLH*).

2.2.2.1 Hachage semi-supervisé linéaire

SSH est aujourd'hui la méthode la plus populaire. Elle vise à répondre à une problématique courante de la recherche d'image dans des bases de données grandissantes : le hachage non-supervisé n'est pas efficace car incapable de saisir les similarité entre les images, et le hachage supervisé est à la fois trop long à entraîner, et fonctionne mal quand les labels sont présents en faible nombre, et/ou contiennent du bruit (des erreurs).

SSH fonctionne en construisant des matrices de covariance [26], c'est à dire des matrices de proximité, entre des données labellisées et non-labellisées. A partir d'un ensemble de données non-labellisées mis à part pour quelques labels de paires de données, *SSH* arrive à construire des fonctions de hachage dépendantes des données, qui permettent un stockage et une recherche rapide. *SSH* s'execute ainsi en temps linéaire.

Pour autant, les performances de *SSH* diminues avec des codes binaires (des valeurs de hachage) de faible longueur, à cause de la faible différence d'un code à l'autre, qui compliquent leur utilisation. C'est pour résoudre ce problème qu'a été proposé le *SSDH*, qui construit les codes binaires en maximisant la différence d'un à l'autre, et qui sur-performe effectivement *SSH* avec des codes binaires courts.

STPH lui vise à répondre, à la fois à l'absence de classement dans l'utilisation des labels par *SSH* et *SSDH* qui compliquent la recherche de similarité entre images, et à l'absence de préservation des données de topologie qui compliquent la recherche de similarité sémantique (c'est à dire la proximité des images en terme de signification plutôt qu'en terme de valeurs de pixels). *STPH* incorpore donc un classement des labels en construisant les matrices de covariance, et se concentre également sur les labels sémantiques pour construire les fonctions de hachage.

2.2.2.2 Hachage semi-supervisé non-linéaire

Les méthodes de hachage semi-supervisé non-linéaires supposent que les fonctions noyaux [27] (donc non-linéaires) sont plus efficaces à saisir les similarités entre les données, surtout pour des images. *LAMP*, dit faiblement-supervisé, partitionne les données en clusters, en conservant une marge de chaque côté de la séparation, pour construire les fonctions de hachage. Comme *LAMP* ne suppose aucune distribution initiale des données, il peut être utilisé pour n'importe quelle base de données d'images, et produit un faible nombre d'erreurs grâce à la marge conservée lors de la séparation des données.

Bootstrap-NSPLH vise à corriger deux limitations de *SSH* : la faible représentation des proximités entre les données à cause de l'usage de fonctions linéaires, et la forte complexité temporale et spatiale pour des grandes quantités de données. *Bootstrap-NSPLH* utilise donc une fonction de hachage non-linéaire, et est également capable de corriger les erreurs accumulées pendant le hachage des données. Cette méthode utilise ainsi des matrices de bien plus faible dimensions que *SSH*, améliorant sa complexité.

2.2.3 Hachage supervisé

En hachage supervisé, sont disponibles des données de labellisation, comme des labels par image, des informations de similarité par paires de données, ou des informations de sémantique d'images, pour permettre d'apprendre de l'espace des données.

On mentionnera :

- Le Hachage supervisé linéaire :
  - Le Hachage Supervisé avec Réduction Binaire (*Supervised Hashing with Binary Reconstruction*).
  - L'Encodage *Boosting* [28] Sensible à la Similarité (*Boosting Similarity-Sensitive Coding*, *BoostSSC*)
  - La Reconstruction Linéaire par Plongement (*Binary Reconstructive Embedding*, *BRE*).
  - Le Hachage à Perte Minimale (*Minimal Loss Hashing*, *MLH*).
  - Le Hachage de Facteurs Latents (*Latent Factor Hashing*, *LFH*).
  - Le Hachage Basé sur l'Analyse Discriminante Linéaire (*Linear Discriminant Analysis-Based Hashing*, *LDAHash*).
- Le Hachage supervisé non-linéaire :
  - Le Hachage Supervisé par Noyau (*Kernel-Based Supervised Hashing*, *KSH*).
  - Le Hachage à Deux Étapes (*Two-Step Hashing*, *TSH*).
  - *FastHash*.

2.2.3.1 Hachage supervisé linéaire

En hachage supervisé linéaire, on essaye généralement d'opérer une réduction binaire sur l'espace des données. Par exemple, si l'espace des données est un ensemble d'images, chaque image est réduite à un faible nombre de couleurs dont la présence de chacune est pondérée pour donner une approximation de l'image initiale. *BoostSSC* apprend de l'espace des données en attribuant des poids aux réductions binaires, et en calculant les distances de Hamming entre les images. *BRE* vise lui à représenter avec le maximum de fidélité la distance entre les données, en attribuant des distances très faibles, voir nulles, aux paires similaires, et de grandes distances aux paires différentes.

Mais du fait de son mode d'apprentissage, *BRE* est largement inefficace sur de grandes bases de données, ayant une complexité logarithmique (en $O(n \log n)$). On lui préférera *MLH*, qui remplace les distances de Hamming par des distances euclidiennes. Succédant à *MLH*, *LFH* emploie une approche d'apprentissage stochastique à variante en temps linéaire, c'est à dire une approche aléatoire linéaire. Enfin, *LDAHash* répond aux problèmes de correspondance et d'accès dans les grandes bases de données rencontrées par les méthodes précédentes en représentant les vecteurs descriptifs (les valeurs numériques des caractéristiques étudiées) dans un espace de Hamming (un espace binaire), sous forme de courts codes binaires.

2.2.3.2 Hachage supervisé non-linéaire

KSH a été le premier modèle proposé pour essayer de résoudre le problème du temps d'entraînement conséquent des algorithmes de hachage supervisé. Comme *TSH*, il utilise une fonction kernel en tant que fonction non-linéaire, tandis que *FastHash* utilise des arbres de décision dits *Boosted* [29], c'est à dire un agrégats d'arbres de faible qualité informationnelle.

Plus précisément, *KSH* calcule un code binaire pour chaque valeur à hacher, en utilisant les informations initiales de similarité ou de dissimilarité de chacune des paires de données, pour minimiser la distance de Hamming entre deux codes de deux valeurs similaires, et la maximiser pour des valeurs différentes. *KSH* obtient ainsi des codes à la fois courts et discriminants (les uns par rapport aux autres). Considérant que l'optimisation permise par *KSH* est en réalité limitée par le nombre restreint de fonctions de hachage non-linéaires réellement optimisables, *TSH* propose de découpler le processus d'apprentissage du processus de hachage. Plus précisément, *TSH* apprend d'abord les codes binaires de chacune des valeurs en utilisant une ou plusieurs fonctions objectifs, puis apprend les fonctions de hachage à partir de ces codes binaires, réalisant donc le processus général en deux étapes entièrement découplées.

C'est face à la lenteur de *TSH* sur des grandes bases de données à haute dimensionnalité que *FastHash* a été proposé. *FastHash* utilise des arbres de décision en tant que fonctions non-linéaires, pour diviser la masse de données en ensembles gérables, et applique ensuite un algorithme d'apprentissage à deux étapes, où il lie les données à leur code binaire respectif, puis les classe en fonction de leur similarité.

De manière générale, grâce à leur force capacité de généralisation à ensemble de situations diverses, les fonctions non-linéaires sur-performent par rapport aux fonctions linéaires. Le hachage supervisé lui est généralement plus flexible que ses concurrents dans des contextes réels, mais peine toujours à résoudre le problème posé par la longueur du temps d'apprentissage des données.

3 Hachage axé sur la sécurité

Le hachage axé sur la sécurité cherche à vérifier l'intégrité des données. Il utilise rarement des tables de hachage, et permet de comparer facilement des données car la valeur de hachage (les données de sortie de la fonction, utilisées comme signature) est bien plus courte que les données d'origine. On identifie deux types de hachages axés sur la sécurité :

- Le hachage sécurisé cryptographiquement, aussi appelé hachage cryptographique.
- Le hachage non sécurisé cryptographiquement, aussi appelé hachage non-cryptographique.

On utilise le hachage axé sur la sécurité principalement dans des implémentations liés à la sécurité informatique, notamment les signatures numériques, qui permettent d'authentifier un message, ou encore les sommes de contrôle qui permettent d'assurer son intégrité.

3.1 Hachage cryptographique

Dans le hachage cryptographique, les fonction de hachage sont non-réversibles (ou le sont très difficilement), la longueur des données d'entrée est arbitraire, mais celle des données de sortie est fixe, et celles-ci doivent changer significativement si même un seul bit des données d'entrée change. Cette méthode se retrouve dans les signatures numériques, le chiffrement à clef publique et l'authentification de messages.

On distingue ainsi trois propriétés de résistance :

1. La résistance à la préimage : Il doit être difficile de trouver une donnée d'entrée en ne connaissant que la donnée de sortie
2. La résistance à la seconde préimage : Pour une clef k, un message m_i, une fonction de hachage h, et un hachage $h(k, m_i)$, c'est à dire en ne possédant qu'un jeu de données de sortie, il est très difficile de trouver un autre message m_j tel que $h(k, m_i) = h(k, m_j)$, c'est à dire une autre donnée d'entrée qui donnerait le même hachage.
3. La résistance aux collisions : Pour deux messages m_i et m_j, tels que $m_i \neq m_j$, une clef k, et une fonction de hachage h, $h(k, m_i) \neq h(k, m_j)$, c'est à dire que deux messages différents ne doivent pas avoir le même hachage.

Attention, on pourrait croire, que la résistance à la seconde préimage et la résistance aux collisions sont la même chose, mais ce n'est pas le cas [30], la résistance à la seconde préimage est plus forte que la résistance aux collisions, car la première implique la deuxième. En effet, dans la résistance à la seconde préimage, l'attaquant récupère un m_i donné, et doit trouver un $m_j \neq m_i$, alors que dans la résistance aux collisions, l'attaquant peut choisir m_i et m_j.

Du fait de ces contraintes sécuritaires, la majorité des implémentations de hachage cryptographique sont au moins en temps exponentiel, en $O(x^n)$, et sont également généralement en complexité spatiale linéaire.

On distingue deux catégories de hachage cryptographique, selon si un secret est utilisé pour le hachage ou non :

1. Hachage cryptographique sans clef (*Keyless cryptographic hashing*). Notamment :
    1. *MD2/4/5/6*
    2. *SHA-1/3/224/256/384/512*
    3. *HAVAL*, *GOST*, *FSB*, *JH*, *ECOH*
    4. *RIPEMD-128/160/320*
2. Hachage cryptographique à clef (*Keyed cryptographic hashing*). Notamment :
    1. *VMAC*, *UMAC*, *PMAC*, *OMAC*, *HMAC*
    2. *Poly1305-AES*, *BLAKE2*

3.1.1 Hachage cryptographique sans clef

Le hachage cryptographique sans clef qualifie toutes les méthodes qui n'ont pas besoin d'une clef sécurisée pour garantir la sécurité du processus de hachage, c'est à dire pour qu'il corresponde aux politiques de sécurité standards du hachage. Parmi les implémentations célèbres, on peut citer les familles des *MD* et *SHA*.

MD, abréviation de *Message Digest*, représente une série de fonctions de hachage que l'on doit à *RSA Data Security Inc*. Les différentes itérations sont :

- *MD1* : Une méthode propriétaire de l'entreprise.
- *MD2* : Une méthode de 1992 destinée à générer une valeur de hachage de 16 bits quelque soit la longueur du message original, avec deux contraintes :
    1. Trouver le message à partir de la valeur de hachage doit demander 2^{128} opérations.
    2. Trouver deux messages différents à partir d'une même valeur de hachage doit demander 2^{64} opérations.

    Mais à cause de vulnérabilités notamment liées à des problèmes de collisions, *MD2* a été déclaré non-sécurisé en 2004.
- *MD4* : Proposée aussi en 1992, elle utilise les mêmes contraintes de collisions que *MD2*, avec une valeur de hachage de 128 octets, et a fait face aux mêmes failles que *MD2*, la rendant elle aussi non-sécurisée.
- *MD5* : Proposée pour répondre aux failles de *MD4*, ajoute une itération [31] à l'opération de hachage, portant le total à 4. En cryptographie, une itération (*round*) est une transformation de base répétée plusieurs fois dans l'algorithme (ici de hachage), qui permet, en découpant l'algorithme, de simplifier son implémentation et son analyse. *MD5* n'a aujourd'hui pas de vulnérabilité connue, mais ne fait plus preuve de l'efficacité nécessaire au hachage dans des environnement *big data*, à quantité massive de données.
- *MD6* : Proposée en 2008, *MD6* est une version améliorée de *MD5*, qui introduit une longueur flexible de la valeur de hachage, de 1 à 512 bits.

La famille des *SHA* est elle initiée par *SHA-0*, souvent ignoré car directement remplacé par *SHA-1*, considéré comme le véritable successeur à *MD4*, en apportant deux changements majeurs :

1. La valeur de hachage passe à 160 bits.
2. Chaque itération consiste en 20 étapes, contre 16 pour *MD4*, et chaque itération est répétée 80 fois, contre 3 pour *MD4*.

On pourrait également mentionner *HAVAL*, un autre algorithme similaire à *MD5*, mais plus flexible par la longueur de la valeur de hachage (128 ou 256), et par le nombre d'itérations, spécifié par l'utilisateur. Ou encore les familles *BLAKE*, *RIPEMD*, *ECOH*, *FSB*, ou *GOST*.

Malgré tout, si ces applications sont encore recommandables pour des applications non-sécuritaires, comme le partitionnement de disques, l'absence de clef les rends vulnérables à de nombreuses attaques, et ne permet pas de prouver mathématiquement leur sûreté.

3.1.2 Hachage cryptographique à clef

Le hachage cryptographique à clef utilise une clef secrète, qui peut permettre de chiffrer la valeur de hachage

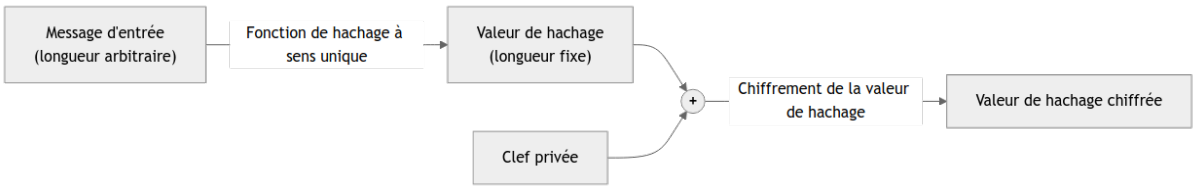

Figure 1: Schéma explicatif du hachage cryptographique à clef

Ce type de hachage est aussi nommé code d'authentification de message, car la valeur de hachage chiffrée, appelée tag d'authentification, va servir de signature pour le message. En effet, le destinataire du message peut, en connaissant la clef privée, calculer de son côté la valeur de hachage chiffrée, et la comparer à celle reçue, pour vérifier l'intégrité du message.

Comme la clef est partagée entre les parties, celle-ci doit être échangée de manière sécurisée. Ces applications de signature sont par exemple utiles dans l'authentification de messages ou de mots de passe.

Parmi les différentes implémentations, on peut citer :

- Le Code d'Authentification de Message à Hachage Universel [32] (*Universal Hashing Message Authentication Code*, *UMAC*), pour les architectures 32-bits, l'un des plus rapides.
- Le Code d'Authentification de Message à Chiffrement par Bloc et à Hachage Universel (*Block Cipher-Based Universal Hashing Message Authentication Code*, *VMAC*), supporte les architectures 32-bits, mais est spécialisé pour les 64-bits, pour lesquelles il atteint des performances importantes.
- Le Code d'Authentification de Message Parallélisé (*Parallelized Message Authentication Code*, *PMAC*), qui utilise aussi le chiffrement par bloc [33] pour permettre de paralléliser le processus de hachage.
- Le Code d'Authentification de Message à Clef Unique (*One-Key Message Authentication Code*, *OMAC*), similaire à *PMAC*.

- Le Code d'Authentification de Message à Hachage de Clefs (*keyed-hash message authentication code*, *HMAC*), qui est un type particulier de code d'authentification de message à deux étapes, utilisant, en plus de la clef, une fonction de hachage cryptographique [34]. Plus précisément, avec *HMAC*, le hash est combiné avec la clef, l'ensemble est haché, et recombiné avec la clef, pour donner le tag d'authentification.

On aurait aussi pu mentionner *Poly1305-AES*, *SipHash*, ou *BLAKE2*.

3.2 Hachage non-cryptographique

Le hachage non-cryptographique partage un objectif similaire au hachage cryptographique, c'est à dire pour un message de longueur arbitraire, retourner une valeur de hachage de longueur fixe, inférieure au message. La grande différence réside dans l'absence de considérations cryptographiques, où il n'y a pas besoin de secret partagé, et où la fonction de hachage est publique, car ce type de hachage est surtout utilisé pour vérifier l'intégrité de données, dans des applications sans grandes contraintes de sécurité.

Comme le processus de hachage ne comporte pas de cryptographie, il permet de se concentrer sur l'optimisation de la vitesse de calcul (la majorité de ses implémentations sont en $O(n)$), et de la détection et gestion des erreurs et des collisions. En plus de leur faible complexité temporelle, la majorité des algorithmes sont en complexité spatiale linéaire, généralement plus économes en mémoire que les algorithmes de hachage cryptographique. Ce type de hachage est très utilisé dans les services avec beaucoup de d'opérations de recherche ou de calcul, parmi lesquels on pourrait citer Twitter, les services DNS, les bases de données…

Les méthodes de hachage non-cryptographiques partages 4 caractéristiques, qui les rendent particulièrement adaptées au traitement de gros volumes de données qui demandent une grande rapidité de calcul :

1. Hachage rapide.
2. Faible probabilité de collisions, ces fonctions essayent de les réduire mais elles restent observables car les fonctions de hachage non-cryptographiques ne suivent pas les mêmes trois règles de sécurité que les fonctions de hachage cryptographiques.
3. Forte probabilité de détection des erreurs.
4. Détection des collisions facilitée.

En plus de ces critères, pour étudier la qualité d'une fonction de hachage non-cryptographique, on s'intéresse également à :

- La distribution des valeurs de hachage, qui doit être uniforme pour éviter la formation d'agrégats qui affecteraient les performances.
- L'effet d'avalanche [35], c'est à dire l'importance du changement dans la valeur de hachage que provoque un changement mineur dans la valeur d'entrée.

La majorité de ces méthodes se base sur la construction de Merkle-Damgard [36], une méthode de construction de fonctions de hachage résistantes aux collisions, qui exécute les étapes suivantes :

1. Utilisation d'un *padding* pour amener la taille des données à un multiple de la taille de données que la fonction de hachage accepte (par exemple un multiple de 512 ou 1024 bits).
2. Séparation de la donnée en un ensemble de blocs de taille égale.
3. Compression de chacun des blocs en une valeur de hachage.
4. Concaténation des valeurs de hachage.
5. Ajout d'une nouvelle valeur de *padding* qui contient de manière encodée la longueur du hachage précédent.

Parmi les implémentations les plus connues, on peut citer :

- Le hachage de *Fowler-Noll-Vo* (*FNV*), une des premières implémentations, destinée au calcul de tables de hachage et de sommes de contrôle [37] (*checksum*), d'implementation simple.
- *Lookup3*, qui génère des valeurs de hachage de 32-bits pour de la recherche dans des tables de hachage, l'une des plus utilisées, notamment par Google ou Oracle
- *SuperFastHash*, inspiré par *FNV* et *Lookup3*, qui atteint des vitesses de calcul très élevées.
- *MurmurHash2*, très populaire dans les projets open-source.
- *MurmurHash3* [38], par exemple utilisé dans l'analyse de sentiments [39] sur Twitter [40].
- *DJBX33A*, spécifiquement développée pour le hachage de chaînes de caractères, très utilisé dans de nombreux langages de programmation et serveurs d'applications, comme *Python*, *PHP 5*, *Apache Tomcat*…
- *BuzHash*, utilise une table de substitution [41], une fonction qui prend m bits d'entrée et renvoie n bits de sortie, et est couramment utilisée pour rajouter de la confusion. La table de substitution sert à remplacer chaque donnée d'entrée par un alias aléatoire, permettant d'appliquer cet algorithme à n'importe quelle distribution de n'importe quelle donnée.
- Trois fonctions populaires de la librairie générale des fonctions de hachage (*General Hash Function Library*) : *DEK* (la plus ancienne et la plus simple, qui utilise le hachage multiplicatif [42]), *BKDR*, et *APartow*.
- *xxHash*, qui supporte les architectures 32 et 64-bits, et est aujourd'hui très populaires dans les bases de données et les jeux vidéos du fait de sa rapidité, uniquement plafonnée par les limites de la RAM.
- La Recherche Approximative [43], très utilisée en analyse forensique, où il est souvent nécessaire de comparer des fichiers pour estimer leur degré de similarité. Les fonctions de hachage cryptographique échouent à cet exercice, en n'étant généralement capables de ne proposer qu'un résultat binaire ("*les signatures sont-elles différentes ou égales ?*"). La Recherche Approximative découle de la méthode dite de Hachage par morceaux déclenché par le context (*Context-Triggered Piecewise Hashing* [44], *CTPH*), aussi appelée Hachage Flou (*Fuzzy Hashing*). Parmi toutes ces méthodes, *MurmurHash2*, *SuperFastHash*, et *Lookup3*, qui, en plus de proposer le meilleur effet d'avalanche, sont également les plus adaptées aux usages généraux. En pratique, on mentionnera des outils comme `ssdeep` [45].

4 Conclusion

Dans ce document, nous avons abordé deux types de fonctions de hachages, respectivement axées sur les données et sur la sécurité, catégorisées de manière hiérarchique. Nous avons vu que le premier visait à accélérer le traitement de données, en étant soit indépendant des données en question, soit dépendant, c'est à dire nécessitant un entraînement préalable sur celles-ci. Nous avons également vu que le deuxième type visait principalement à sécuriser des échanges, au travers de fonctions cryptographiques ou non, c'est à dire impliquant ou non une irréversibilité de l'opération pour dissimuler une donnée. Enfin, nous avons abordé au cours du texte les différentes implémentations pratiques de chacune des méthodes, ainsi que leurs avantages et inconvénients, et leur complexité spatiale et temporelle.

Pour plus d'informations, veuillez vous référer à l'article original, *Hashing Techniques: A Survey and Taxonomy*, par *LIANHUA CHI* et *XINGQUAN ZHU*, très complet et fournit. Malgré tout, on regrettera dans cet article original des erreurs, allant de la coquille au contre-sens, comme les confusions entre *bits* et *bytes*, ou la répétition d'un même terme au lieu du terme puis de son antonyme, changeant le sens de la phrase, comme *unsupervised term* page 18.

Vous trouverez ci-dessous un schéma récapitulatif des différentes méthodes abordées, et de leur emplacement dans notre taxonomie.

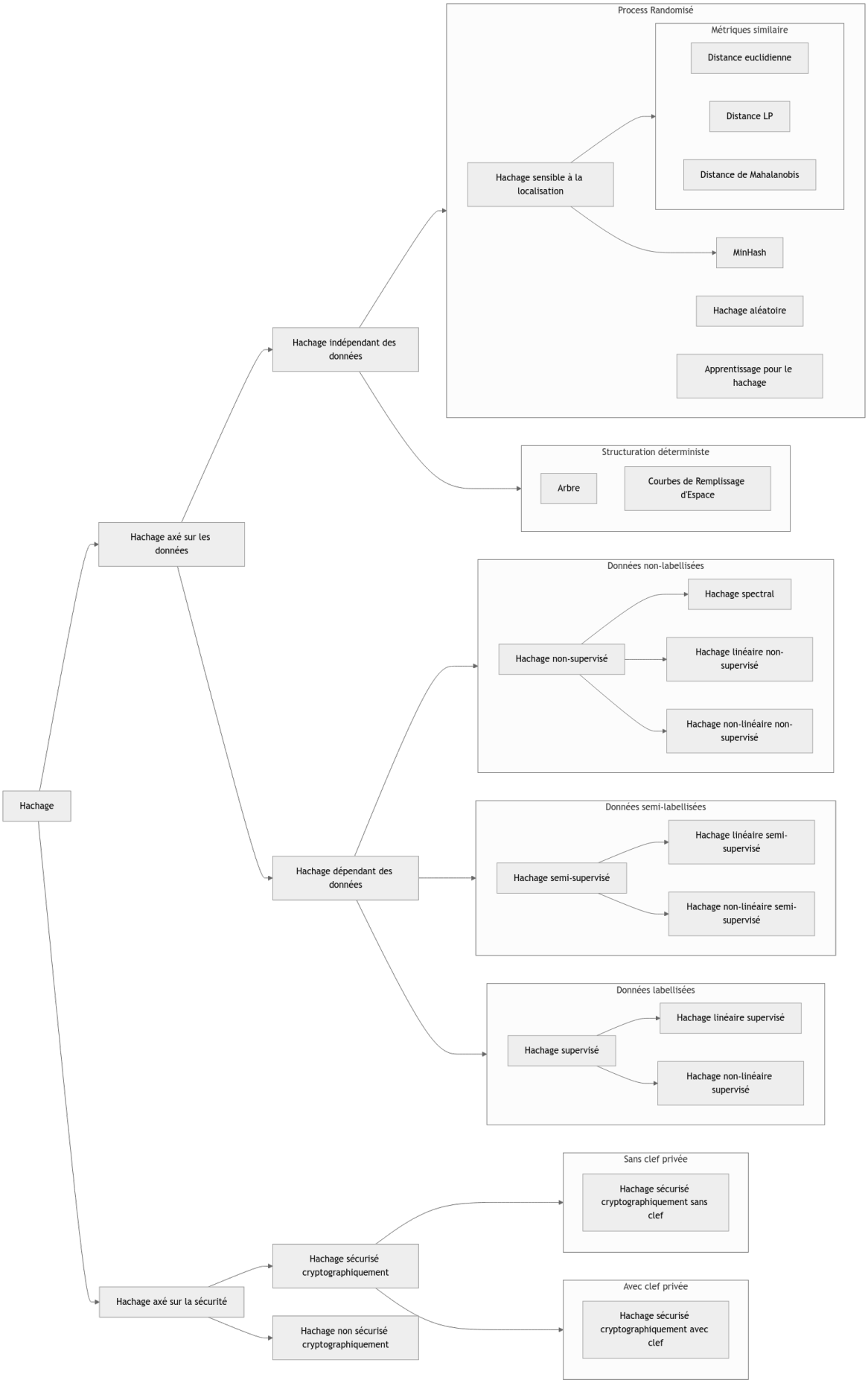

Figure 2: Schéma récapitulatif des différentes méthodes de hachage

5 Références

[1] Stephanie, “Jaccard Index / Similarity Coefficient”, 2016. Available: https://www.statisticshowto.com/jaccard-index/.

[2] “Hamming Distance - An overview | ScienceDirect Topics”. Available: https://www.sciencedirect.com/topics/engineering/hamming-distance.

[3] A. I. Maverick, “A Comprehensive Guide to Latent Space”, 2023. Available: https://samanemami.medium.com/a-comprehensive-guide-to-latent-space-9ae7f72bdb2f.

[4] “Chapitre 3 - Espaces Euclidiens”. Available: https://licence-math.univ-lyon1.fr/lib/exe/fetch.php?media=p12:algiv:chapitre3_algebreiv.pdf.

[5] “Loterre: Mathématiques: espace pseudo-euclidien”, 2023. Available: https://skosmos.loterre.fr/PSR/fr/page/-MKZ7N5M3-5.

[6] “Géométries euclidienne et non euclidiennes”. Available: https://culturemath.ens.fr/thematiques/superieur/geometries-euclidienne-et-non-euclidiennes.

[7] L. Todjihounde, “Calcul différentiel 2e édition”, Éditions Cépaduès - Google Books, 2024. Available: https://books.google.ca/books?id=E-P9kDL0XQoC&pg=PA276&redir_esc=y#v=onepage&q&f=false.

[8] Y. Freund and R. E. Schapire, “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”, Journal of Computer and System Sciences, vol. 55, no. 1, pp. 119-139, 1997. doi: 10.1006/jcss.1997.1504.

[9] “AdaBoost”. Available: https://en.wikipedia.org/wiki/AdaBoost.

[10] R. A. Finkel and J. L. Bentley, “Quad trees a data structure for retrieval on composite keys”, Acta Informatica, vol. 4, no. 1, pp. 1-9, 1974. doi: 10.1007/bf00288933.

[11] “Quadtree”. Available: https://en.wikipedia.org/wiki/Quadtree.

[12] E. W. Weisstein, “Hilbert Curve”. Available: https://mathworld.wolfram.com/HilbertCurve.html.

[13] H. Khelif, “Sur les courbes de remplissage”, 2024. Available: http://images.math.cnrs.fr/Sur-les-courbes-de-remplissage.

[14] C.-T. Zhang, R. Zhang, and H.-Y. Ou, “The Z curve database: a graphic representation of genome sequences”, Bioinformatics, vol. 19, no. 5, pp. 593-599, 2003. doi: 10.1093/bioinformatics/btg041.

[15] “Z curve”. Available: https://en.wikipedia.org/wiki/Z_curve.

[16] “Analyse en composantes principales”. Available: https://spss.espaceweb.usherbrooke.ca/analyse-en-composantes-principales-2/.

[17] W. A. Wassam, “Statistical Mechanics”, in Elsevier eBooks, pp. 749-819, 2003. doi: 10.1016/b0-12-227410-5/00727-4.

[18] M. Abdolali and N. Gillis, “Beyond linear subspace clustering: A comparative study of nonlinear manifold clustering algorithms”, Computer Science Review, vol. 42, p. 100435, 2021. doi: 10.1016/j.cosrev.2021.100435.

[19] D. Pnevmatikatos and A. Arelakis, “Variable-Length Hashing for Exact Pattern Matching”, in 2006 International Conference on Field Programmable Logic and Applications, pp. 1-6, 2006. doi: 10.1109/fpl.2006.311244.

[20] Y. Lin, R. Jin, D. Cai, S. Yan, and X. Li, “Compressed Hashing”, in IEEE, 2013. doi: 10.1109/cvpr.2013.64.

[21] J. Mairal, F. Bach, J. Ponce, and G. Sapiro, “Online Learning for Matrix Factorization and Sparse Coding”, Journal of Machine Learning Research, vol. 11, pp. 19-60, 2010. Available: https://www.jmlr.org/papers/volume11/mairal10a/mairal10a.pdf.

[22] “Codage parcimonieux”. Available: https://fr.wikipedia.org/wiki/Codage_parcimonieux.

[23] “Compressive Sensing Resources”. Available: https://dsp.rice.edu/cs/.

[24] “Acquisition comprimée”. Available: https://fr.wikipedia.org/wiki/Acquisition_comprim%C3%A9e.

[25] “Noyaux (Statistiques)”. Available: https://fr.wikipedia.org/wiki/Noyau_(statistiques).

[26] “Vecteurs gaussiens”. Available: https://www.math.univ-paris13.fr/~tournier/fichiers/agreg/statistiques.pdf.

[27] “Astuce du noyau”. Available: https://fr.wikipedia.org/wiki/Astuce_du_noyau.

[28] “Boosting”. Available: https://fr.wikipedia.org/wiki/Boosting.

[29] K. Woodruff, “Introduction to boosted decision trees”, 2017. Available: https://indico.fnal.gov/event/15356/contributions/31377/attachments/19671/24560/DecisionTrees.pdf.

[30] “Second pre-image resistance vs Collision resistance”. Available: https://crypto.stackexchange.com/questions/20997/second-pre-image-resistance-vs-collision-resistance.

[31] JP. Aumasson, “Serious Cryptography”, No Starch Press - Google Books, 2017. Available: https://books.google.ca/books?id=W1v6DwAAQBAJ&pg=PA56&redir_esc=y#v=onepage&q&f=false.

[32] “Hachage Universel”. Available: https://fr.wikipedia.org/wiki/Hachage_universel.

[33] “Self-Study Course in Block Cipher Cryptanalysis”. Available: https://www.schneier.com/academic/archives/2000/01/self-study_course_in.html.

[34] “Fonction de hachage cryptographique”. Available: https://fr.wikipedia.org/wiki/Fonction_de_hachage_cryptographique.

[35] “Effet avalanche”. Available: https://fr.wikipedia.org/wiki/Effet_avalanche.

[36] “Merkle-Damgard Scheme in Cryptography”. Available: https://www.geeksforgeeks.org/merkle-damgard-scheme-in-cryptography/.

[37] “Somme de contrôle”. Available: https://fr.wikipedia.org/wiki/Somme_de_contr%C3%B4le.

[38] aappleby, “smhasher: The home for the MurmurHash family of hash functions”, 2015. Available: https://github.com/aappleby/smhasher.

[39] D. Boullier and A. Lohard, “Opinion mining et Sentiment analysis”, OpenEdition Press, 2012. doi: 10.4000/books.oep.198.

[40] N. Silva, E. Hruschka, and E. Rafael, “Biocom Usp: Tweet Sentiment Analysis with Adaptive Boosting Ensemble”, 2014. Available: https://aclanthology.org/S14-2017.pdf.

[41] “S-Box”. Available: https://fr.wikipedia.org/wiki/S-Box.

[42] “CSE 241 - Algorithms and Data Structures - Choosing Hash Functions”. Available: https://classes.engineering.wustl.edu/cse241/handouts/hash-functions.pdf.

[43] “Recherche approximative”. Available: https://fr.wikipedia.org/wiki/Recherche_approximative.

[44] C. Zheng, X. Li, Q. Liu, Y. Sun, and B. Fang, “Hashing Incomplete and Unordered Network Streams”, in HAL - INRIA, pp. 199-224, 2018. doi: 10.1007/978-3-319-99277-8_12. Available: https://inria.hal.science/hal-01988840.

[45] “ssdeep - Fuzzy hashing program,” ssdeep-project.github.io. Available : https://ssdeep-project.github.io/ssdeep/index.html