École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. (33)2-47-36-14-14
Fax (33)2-47-36-14-22
www.polytech.univ-tours.fr

Parcours des écoles d'Ingénieurs Polytech

Rapport de projet S4

Algorithme de détection de duplicata sur des vidéos en flux live

Auteur(s)

Léo Lefaucheux
[leo.lefaucheux@etu.univ-tours.fr]
Thomas Fournier
[thomas.fournier@etu.univ-tours.fr]

Encadrant(s)

Mathieu Delalandre
[mathieu.delalandre@univ-tours.fr]

Polytech Tours
Département Informatique

Version du 15 avril 2018

Table des matières

Table des figures

Introduction

Le projet de détection de duplicata vidéo nous a interpellé car l'algorithme se trouvant derrière une telle fonction nous intéressait. De plus, nous sommes familiers des milieux de la vidéo et du streaming ce qui nous a d'autant plus poussé à choisir ce sujet. En effet au delà du fait de repérer une copie, la description du projet indiquait qu'il fallait réaliser cette tâche sur un flux en direct.

Ne sachant que trop peu les méthodes et fonctions existantes, ce qu'on aurait à créer et le temps que prendrait les étapes, nous avons décidé d'avancer pas à pas en commençant par les étapes cruciales afin d'avancer le plus possible dans le projet sans se fixer d'objectifs non ou difficilement réalisables.

Concernant le langage de programmation, nous avons dès le départ choisi de travailler avec Python, étant un langage sur lequel nous avions tous deux des connaissances grâce aux cours que nous avions eu mais également parce qu'il est simple d'utilisation et qu'il contient de nombreuses librairies pratiques et faciles d'accès.

Ainsi nous avons débuté notre travail, en commençant par réfléchir à la manière de comparer deux images, étant la première tâche qu'il nous faudrait réaliser. Celà fait, nous avons cherché comment réaliser une base de données qui servirait de support à la détection de duplicata. Une fois la détection de copie vidéo fonctionnelle en prenant des vidéos complètes en notre possession, nous nous sommes attelés au caractère "live" du projet, nous avons donc fait en sorte de pouvoir capturer un flux direct et le traiter au fur et à mesure. Nous détaillerons toutes ces étapes et les solutions trouvées par la suite.

Chapitre 1

Comparaison d'images

Si notre problématique est de détecter des duplicatas sur des vidéos, la première étape était d'arriver à comparer des images. En effet, une vidéo est une suite d'images accompagnées d'une bande sonore.

1.1 La fonction hash

Afin de les comparer entre elles, nous donnons à chaque image un code hexadecimal de 16 caractères qui leur est propre et qui pourrait s'apparenter à leur empreinte. Nous appelons ces codes des hash car ils s'obtiennent avec une "hash function" ou fonction de hachage en français. Ceux-ci sont liés aux images, c'est-à dire qu'en modifiant une image, son hash sera aussi modifié.

La fonction de hash que nous utilisons vient d'une librairie Python du nom de imagehash [1] de Johannes Buchner. Elle est basée sur PIL [2], une librairie dédiée à la manipulation des images avec Python créée par Fredrik Lundh. Il existe différentes fonction de imagehash permettant de calculer ces empreintes.

Elles utilisent différentes méthodes qui ont différents avantages et inconvénients. Les trois principales sont aHash, pHash et dHash.

- La fonction aHash qui signifie "Average Hash" est rapide mais peu précise.
- La fonction pHash pour "Perception Hash" est la plus précise mais elle est beaucoup plus lente, elle utilise une méthode appelée "Discrete Cosine Transform".
- Enfin, la fonction dHash qui signifie "Difference Hash" est une amélioration de la méthode aHash. Elle est aussi rapide et beaucoup plus précise.

Notre projet consiste à travailler en direct sur des vidéos. La rapidité de notre algorithme est donc primordiale. C'est pourquoi nous avons chosi d'utiliser la méthode du "Difference Hash". Le code obtenu avec cette fonction correspond à la différence de luminosité entre les zones de l'image. Pour cela, il faut tranformer l'image pour qu'elle soit plus petite et qu'elle comporte moins de couleurs. Voici une explication détaillée de son fonctionnement accompagnée d'un exemple 1.1 :

1. On commence par passer l'image en niveaux de gris afin de réduire le nombre de couleurs. Le niveau de gris de chaque pixel correspond à sa luminosité sur une échelle du blanc au noir.
2. Il faut aussi réduire la taille de l'image, on la recadre afin qu'elle ne fasse plus que 9x8 pixels soit 72 pixels en tout.
3. Enfin il faut créer le code. Pour cela, on assigne un bit à chaque pixel sauf à ceux de la dernière colonne. Ce qui fait donc 8x8 soit 64 bits. On donne la valeur 1 aux pixels plus clairs que leur voisin de droite et 0 aux autres. On écrit les bits de gauche à droite, puis de haut en bas. Une fois converti en hexadécimal, le hash fait bien 16 caractères.

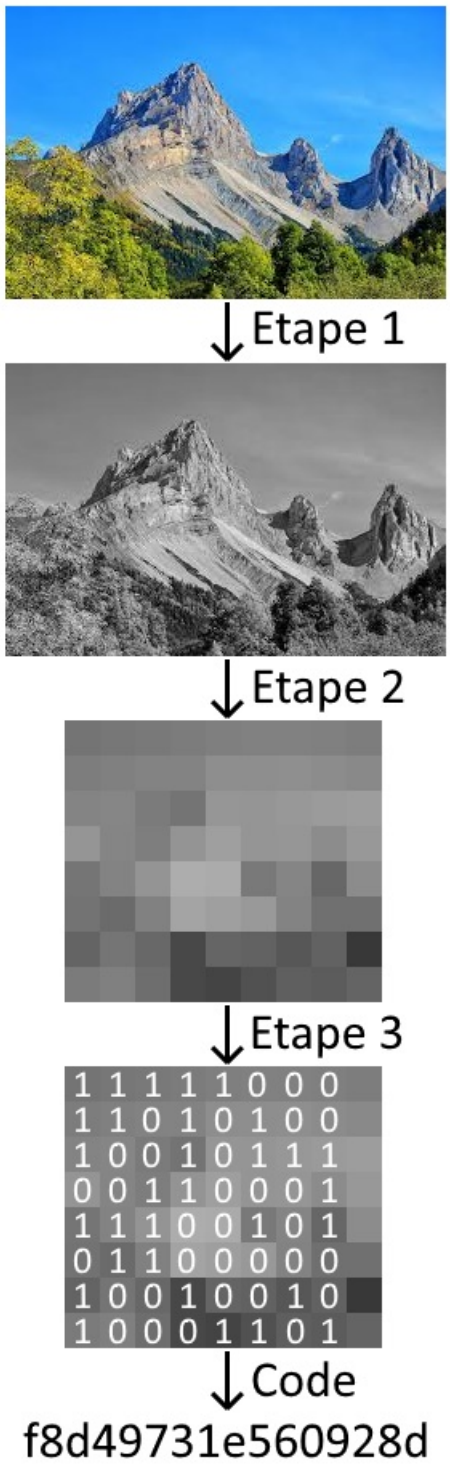

FIGURE 1.1 – Schéma du fonctionnement de la fonction dHash.

Concernant les performances, cette fonction est assez rapide. Toutefois la vitesse d'exécution dépend de la taille de l'image. Nous avons donc testé avec plusieurs tailles d'images que nous avons répertoriées dans le tableau 1.1. Pour qu'il soit efficace notre programme doit prendre moins d'une seconde pour traiter une image.

Taille de l'image	Temps d'exécution
275*183	0.024s
960*552	0.034s
3648*2736	0.358s

TABLE 1.1 – Tableau des durées d'exécution de la fonction par rapport à la taille de l'image.

On remarque que le temps d'exécution augmente beaucoup pour les images de grandes tailles. Cependant il reste relativement bas et largement en dessous d'une seconde ce qui nous laisse de la marge pour le reste de nos fonctions. La méthode dHash est donc bien adaptée à nos objectifs.

1.2 Recherche de correspondance de hash

Pour comparer les images, nous comparons donc les hash. Pour cela nous avons donc créé une base de données contenant les hash.

Ainsi lorsque nous voulons tester si notre image appartient à la base de données, nous commençons par calculer son hash. Puis nous vérifions si ce celui-ci est présent dans la base de données.

L'objectif est de le vérifier rapidement pour pouvoir vérifier une image par seconde. Pour cela nous avons mis nos hash en clé dans notre base de données et non en valeur. Grâce à cette méthode, au lieu de tester toutes les entrées pour voir si la valeur est notre hash recherché, ce qui prend du temps. Nous regardons juste si une entrée existe au hash recherché.

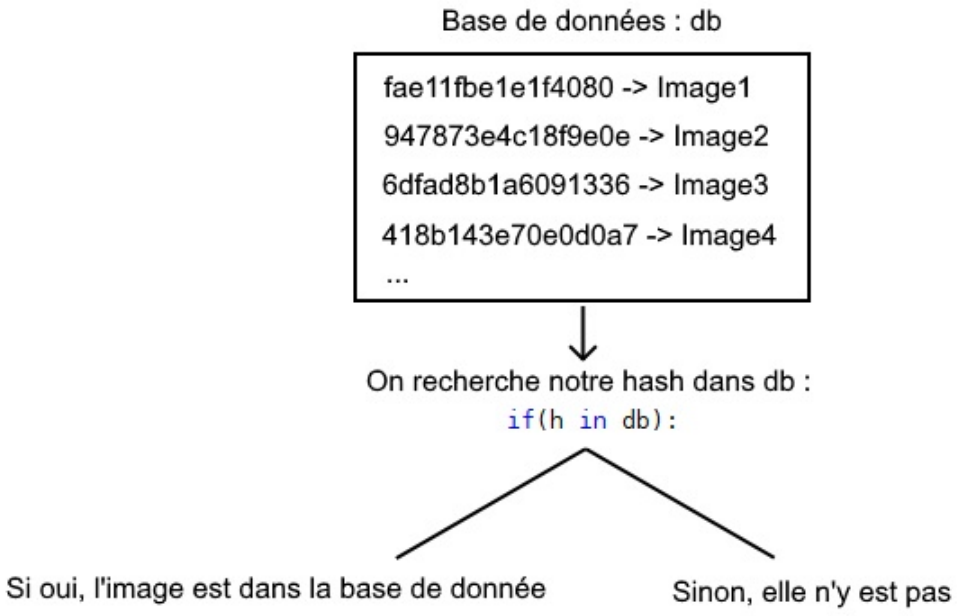

FIGURE 1.2 – Schéma de la recherche d'une image dans la base de données.

1.3 Le hamming entre des hash

Cette technique permet de détecter si une image ou une version un peu modifiée de celle-ci est dans la base de données. Néanmoins nous avons vu qu'en modifiant plus une image, celle-ci change très légèrement de hash 2.3.

FIGURE 1.3 – Deux images à différentes résolutions ont des hash légèrement différents.

Ainsi, si quelqu'un diffuse une vidéo dont il ne détient pas les droits mais avec une basse résolution, notre algorithme ne le détecterait pas.

Pour pallier à ce problème nous avons utilisé plusieurs fichiers de base de données que nous utilisons pour obtenir une liste de candidats (cf section 2.3). Ces candidats sont des images qui ont des hash ressemblant à celui de notre image cible.

Cela nous évite encore une fois de parcourir toute la base de donnée pour vérifier si un code ressemble à celui que l'on recherche. À la place nous étudions un petit échantillon d'images : notre liste de candidats.

Pour chaque candidat, nous calculons la différence entre son hash et celui de la cible. Pour cela, nous utilisons la distance de Hamming, qui compte le nombre de caractères différents.

Si nous détectons une image ayant une distance de Hamming de 5 ou moins, alors nous considérons que l'image est dans la base de données.

Pour reprendre l'exemple de la figure 2.3, la distance de Hamming entre les deux codes est de 1. Les deux images sont donc les mêmes.

Chapitre 2

Réalisation de la base de données

Afin de pouvoir comparer des images et repérer les duplicata d'une vidéo il est nécessaire d'avoir une base de données comprenant les images des vidéos que l'on cherche à protéger ou dont on cherche des copies.

2.1 Fragmentation de vidéos en images

Nous avions donc tout d'abord besoin de trouver comment découper une vidéo en images puisque nous utiliserons une comparaison image par image comme énoncé plus haut. Pour ce faire nous avons utilisé le logiciel de traitement audio/vidéo FFmpeg [3] qui nous permet au travers d'une simple commande, indiquant le dossier de la vidéo et le dossier où enregister les images, à effectuer dans une console, de découper notre vidéo en images en choisissant un nombre d'images par seconde. Le schéma 2.1 illustre cette commande.

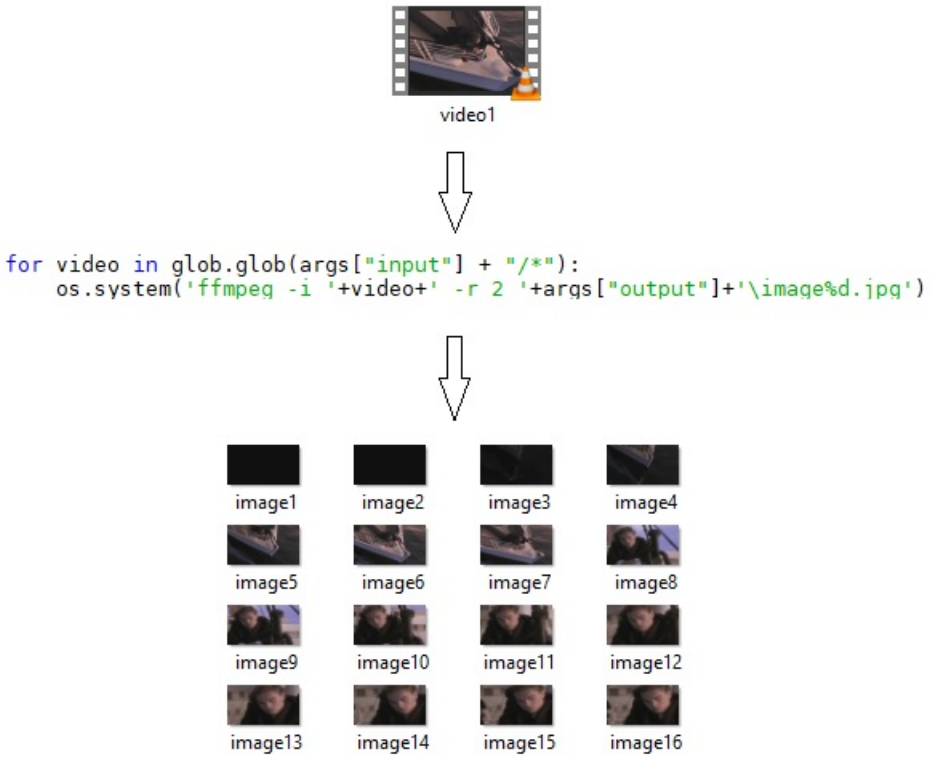

FIGURE 2.1 – Utilisation de FFmpeg

2.2 Altérations et modifications d'images

Dans un second temps, nous avons réfléchi aux différentes possibilités concernant les copies de vidéo. En effet, des copies conformes peuvent être utilisées, cependant dans le cadre d'utilisation frauduleuse d'une vidéo, il est très probable que cette dernière soit un tant soit peu différente de la vidéo d'origine, que ce soit pour éviter d'être repérée ou tout simplement à cause d'une qualité moindre ou d'une dimension différente. Ainsi nous avons décidé d'altérer nous même les images issues de nos différentes vidéos avant de créer leur hash pour élargir notre champ d'action. Nous avons choisi de créer des copies dans un autre dossier, une en négatif, une en symétrie horizontale, une légèrement plus grande et une légèrement plus petite. Pour cela nous utilisons la librairie PIL qui contient des fonctions toutes faites. Cela nous octroie un meilleur panel de comparaison pour les images des vidéos à comparer. On joint à ces images altérées une copie originale puisque nous créerons la base de données à partir de ce dossier. Le schéma 2.2 illustre donc le fonctionnement de cette fonction qui se trouve dans le fichier "gather_video". Une fois ces copies créées nous avons donc le dossier complet d'images qui nous permettra de créer notre base de données de hash.

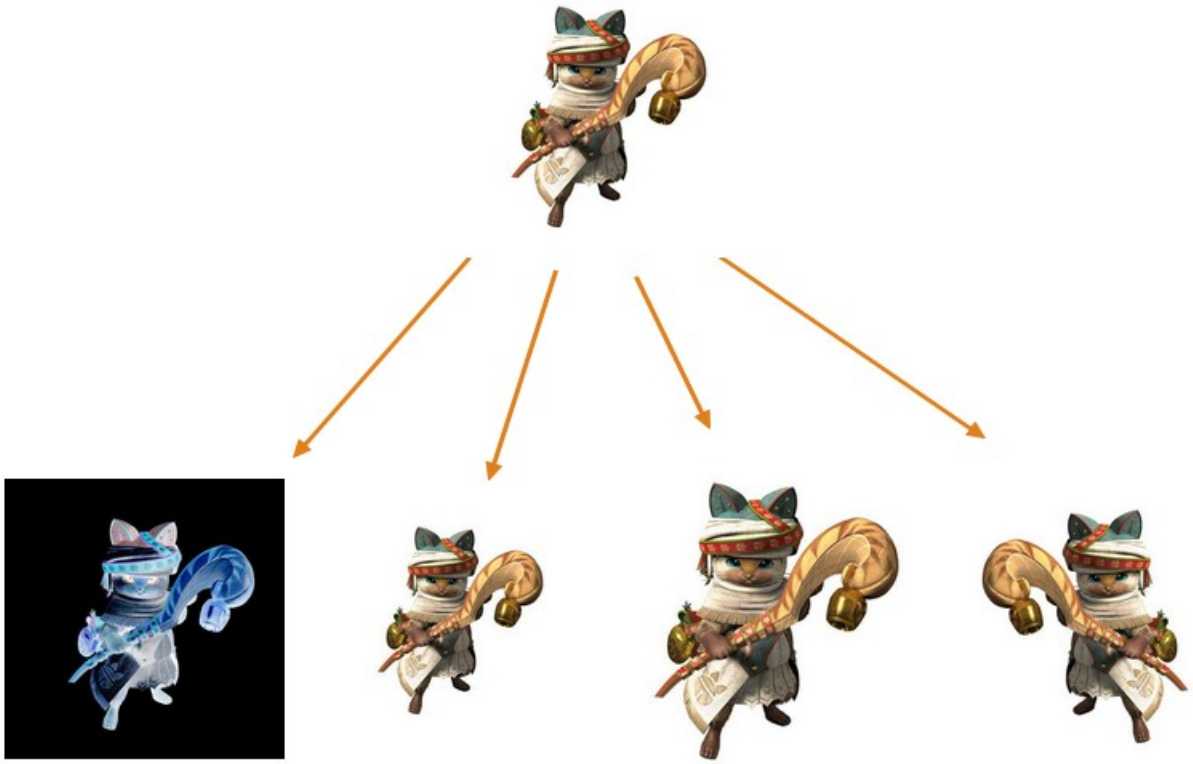

FIGURE 2.2 – Fonctionnement de gather.py

2.3 Création des fichiers contenant les hash

Avec le dossier d'images généré comme indiqué ci-dessus, il nous est maintenant possible de créer les hash de chaque image et de les inscrire dans notre document shelve qui est un type de base de données. Ainsi nous aurons une liste des tous les hash de nos images, ce qui nous permettra de supprimer les images et même les vidéos de nos dossiers pour libérer de la place puisque nous travaillerons seulement avec ces derniers.

Pour les rentrer dans notre base de données, comme expliqué dans la section ?? nous renseignons le hash en clé et le nom de la vidéo dont l'image est tirée en valeur. Grâce à cela, nous avons donc un fichier shelve qui contient toutes les images.

Il faut maintenant un moyen de récupérer la liste des candidats comme expliqué dans la section 1.3. Pour cela, nous avons utilisé une méthode de notre invention qui utilise 4 fichiers de base de données supplémentaires. Lorsque nous calculons un hash, nous en extrayons ses 4 quarts pour les rentrer dans des fichiers shelve dédiés. Comme pour la base de données principale, les quarts de code sont rentrés en clés. Les valeurs associées sont les hash contenant le quart à l'emplacement lié au fichier. Voici un exemple avec quelques empreintes.

Nous faisons tout cela dans un fichier nommé "main_video". Celui-ci utilise la méthode d'altération du fichier "gather_video" de la section 2.2 afin de créer les images modifiées avant de les rentrer dans la base de données.

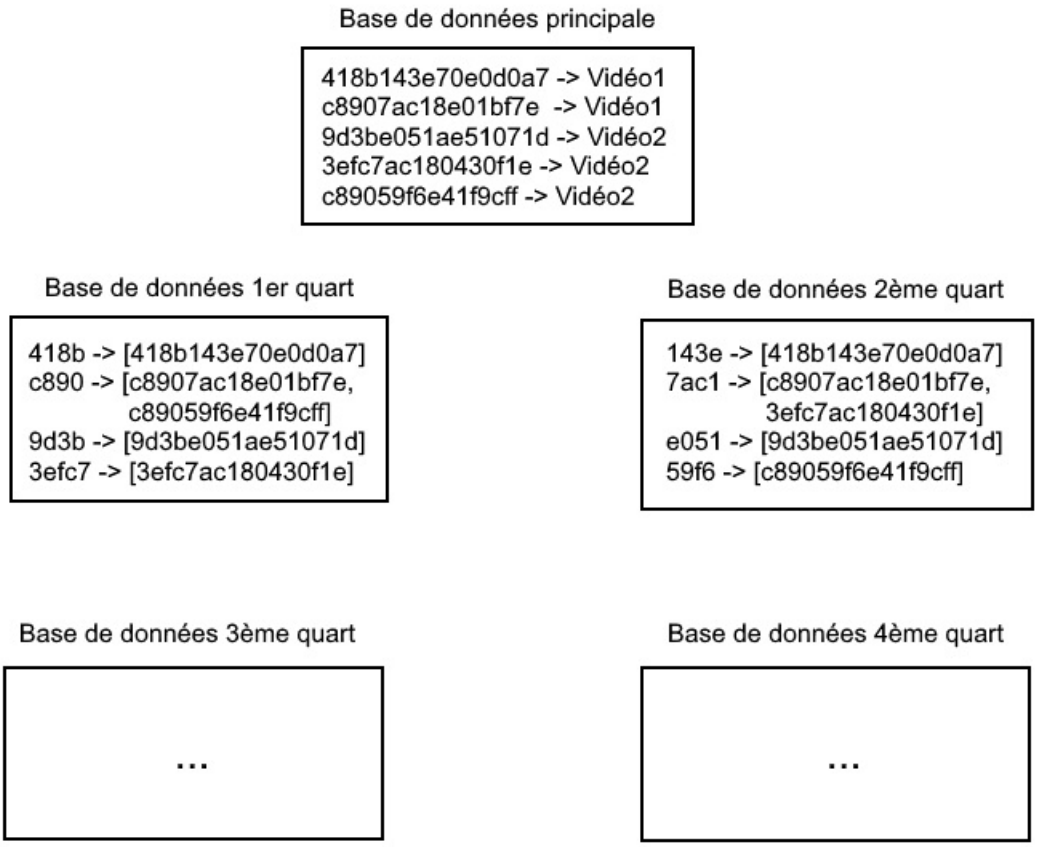

FIGURE 2.3 – Schéma des différentes bases de données.

Ainsi, lorsque nous vérifions la présence d'une image dans la base de données, nous commençons par chercher si son hash est présent dans le fichier principal. S'il l'est alors il est clair que l'image correspond à un duplicata. Sinon, nous coupons ce code en 4 pour chercher dans les 4 bases de données si certaines images partagent un quart en commun avec notre cible. Cela nous crée donc une liste de candidats que nous pouvons étudier avec la distance de Hamming expliquée dans la section 1.3.

Chapitre 3

Transposition au caractère live

Une fois capables de comparer deux vidéos complètes en notre possession nous nous sommes attelés à faire de même avec un flux vidéo en direct. Il nous fallait donc récupérer les images du flux et comparer leur hash avec ceux de notre base de données en direct.

3.1 Capture d'image d'un flux direct

Tout d'abord nous avons réfléchi à la manière de récupérer les images du flux live étant donné que nous ne pouvions plus utiliser la fonction FFmpeg comme précédemment étant donné que nous ne possédons pas la vidéo complète puisque diffusée en direct. Nous avions d'abord pensé à utiliser un logiciel de capture d'écran qui devrait être lancé séparément de notre programme mais nous avons finalement trouvé une fonction python nous permettant de le faire. En effet, la librairie PIL contient une fonction de capture d'écran appelée ImageGrab. Celle-ci nous permet de capturer l'écran dans son entièreté ou partiellement si l'on y ajoute l'argument bbox. Cependant cet argument demande de savoir les coordonnées des pixels des coins haut gauche et bas droit de la zone à capturer. Ces informations sont assez facilement trouvables cependant cela nécessite une manipulation manuelle en amont. La capture partielle peut être utile si l'on veut analyser deux flux à la fois ou encore si l'on sait qu'au sein même d'un flux quelqu'un est suceptible de diffuser une vidéo ne lui appartenant pas (par exemple) sur une zone plus restreinte. Ainsi si quelqu'un diffuse du contenu sous copyright sur un stream en ayant sa caméra sur une partie de l'écran, on pourra définir une zone plus précise à comparer avec la base de données, en effet, notre programme ne détecterait pas l'image comme une copie si une autre image lui est superposée ou que l'image se trouve au sein d'une image plus grande comme l'illustrent les figures 3.1.

FIGURE 3.1 – Exemple de cas nécessitant un recadrage

3.2 Comparaison de hash en continu

Lorsque nous récupérons une capture de la vidéo, nous ne l'enregistrons pas sur l'ordinateur. En effet on peut créer un objet Image et l'utiliser directement pour avoir le hash. Cela permet à notre programme d'économiser un temps précieux.

Ainsi, l'image est capturée puis le hash créé immédiatement. Ensuite nous testons si ce hash correspond à une vidéo en utilisant la méthode expliquée dans la section 1.3. Dès que ce traitement est terminé nous sommes prêts à passer à la capture suivante.

La comparaison en continu des images les unes après les autres permet de détecter en direct une vidéo qui n'est pas libre de droit. Cela est possible grâce à la rapidité des fonctions que nous utilisons. Nous avons d'abord fixé l'objectif à une image à tester par seconde. Mais pour plus de précision, nous pourrions augmenter ce nombre dans la limite de la rapidité d'exécution.

3.3 Critère d'arrêt

Une fois notre programme lancé, celui-ci compare en continu des captures d'écran aux images de notre base de données. Il faut donc prévoir un critère d'arrêt pour que celui-ci ne tourne pas indéfiniment.

Nous avons choisi de l'arrêter lorsqu'il détecte qu'une vidéo présente dans notre base de données a été utilisée sur le stream. Pour cela, nous avons considéré que si 30 images avaient été trouvées dans la base de données, le programme s'arrêtait car on considérait que le stream n'était pas autorisé. Afin d'indiquer que le programme a terminé, nous faisons apparaître une pop-up (Figure 3.2) qui indique que le programme a détecté une vidéo non utilisable. Évidemment, ce système de fin pourrait être amélioré.

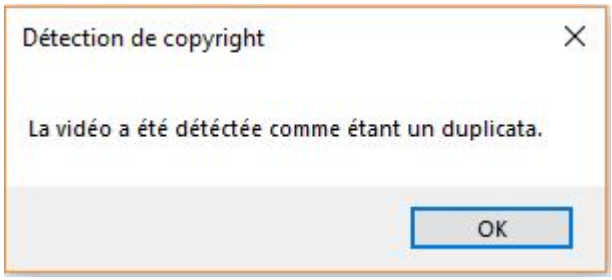

FIGURE 3.2 – Pop-up indiquant la détection d'un duplicata.

Nous avons aussi prévu que le programme s'arrête seul au bout d'un certain temps même si l'utilisateur peut l'arrêter lui-même.

Conclusion

Schéma explicatif du rendu

Le schéma 3.3 résume le fonctionnement de nos différents fichiers python et montre sur quels éléments et à quels moments ils interviennent.

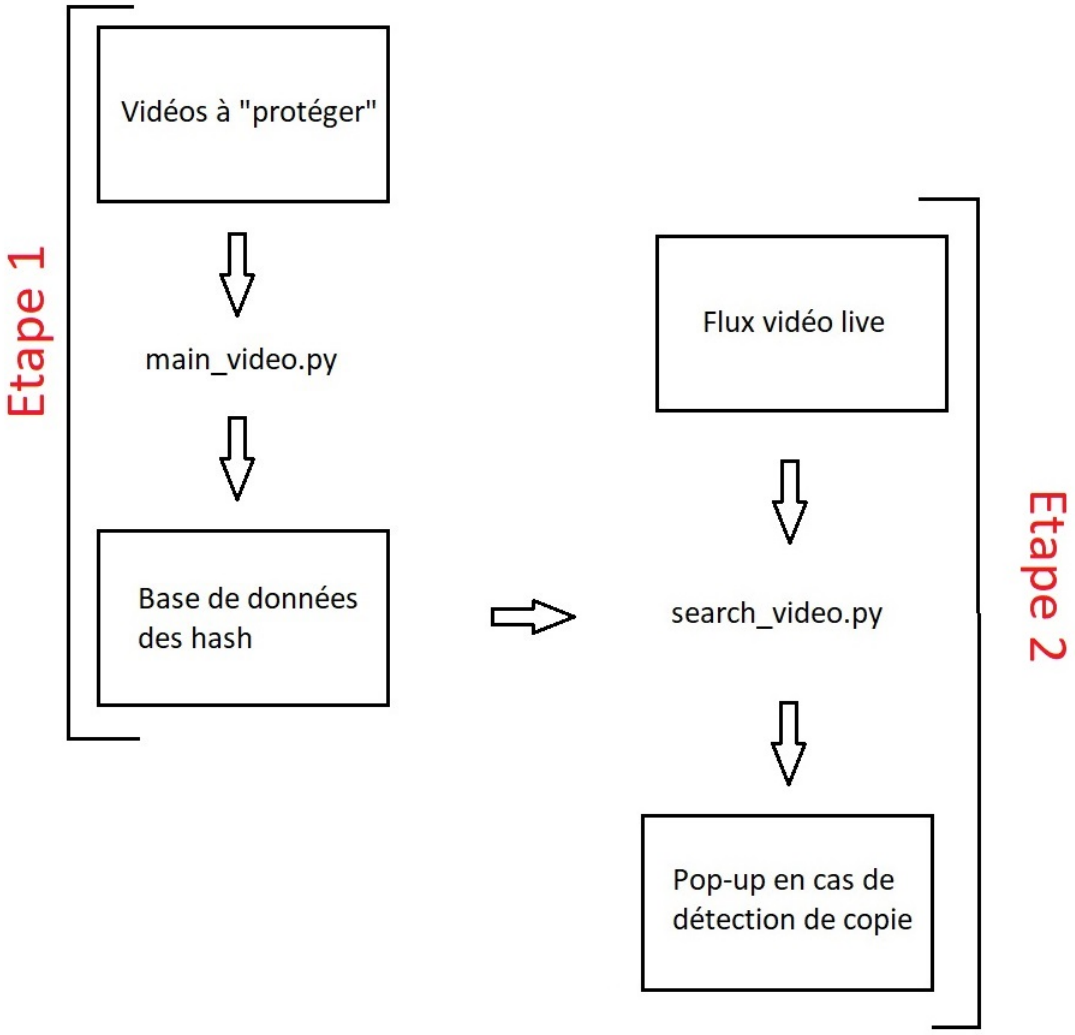

FIGURE 3.3 – Schéma récapitulatif

Le programme final

Nous avons réalisé de nombreux tests afin de déterminer les forces, les faiblesses et les limites de notre programme. Notre problématique était de détecter un duplicata en direct, ce à quoi nous pouvons répondre que notre programme en est maintenant capable. En effet, si nous diffusons sur notre écran une vidéo qui est dans la base de données, en moins de 40s une pop-up apparaîtra pour indiquer que cette vidéo est illégale.

Par exemple avec la vidéo de la figure 3.4, le programme met 36s pour détecter 30 images de la base de données. Le logiciel testant une image par seconde, cela signifie que 6 images n'ont pas eu de correspondance dans la base de données. En augmentant le nombre d'images par seconde injectées dans notre base de données lors de la fragmentation des vidéos, ce chiffre serait significativement réduit.

FIGURE 3.4 – Vidéo présente dans la base données, que notre programme détecte en 36s.

Mais comme nous l'avons évoqué dans la section 2.2, le streameur pourrait diffuser une vidéo qui soit légèrement modifiée, soit pour déjouer les programmes de détection comme le notre ou soit parce qu'il s'en est procuré une version légèrement différente. Nous avons donc testé différentes modifications de la vidéo 3.4. Voici ces quelques exemples ainsi que les résultats correspondants.

D'après ces tests, nous pouvons remarqué que de nombreuses modifications sont prises en charge par notre programme. Quand à celles qui ne le sont pas, il suffirait d'ajouter de nouvelles altérations au fichier gather_video permettant à notre base de données d'être plus complète. Cela la rend évolutive et permet à notre programme de s'améliorer au fur et à mesure.

Si notre programme se doit d'être précis dans la détection, il doit aussi être efficace et rapide. En effet nous devons tester en direct, ce qui nous oblige à traiter une image en moins d'une seconde si nous souhaitons tester une image par seconde. Plusieurs choses peuvent influer sur la rapidité d'exécution :

— La taille de la vidéo à tester, plus l'image est grande et plus la capture et le hash seront longs à effectuer.
— La puissance de l'ordinateur sur lequel le programme tourne.
— La taille de la base de données.

Pour les deux premiers points, il n'y a pas réellement de problème. En effet nos tests ont été effectués sur des images assez grandes et les performances étaient très bonnes. Tant que le

FIGURE 3.5 – La vidéo avec une webcam dans un coin est bien détectée comme un duplicata.

FIGURE 3.6 – La vidéo avec sa saturation légèrement baissée est détectée comme duplicata.

FIGURE 3.7 – La vidéo avec de légères bandes noires est détectée comme duplicata

FIGURE 3.8 – Mais avec de grandes bandes noires, elle n'est plus détectée.

stream, n'est pas en Ultra HD, la capture et la création de hash devraient prendre moins de 0.1s. De plus, les ordinateurs sur lesquels nous avons effectué nos tests ne sont pas extrêmement puissants.

Pour la base de données, nous avons testé différentes tailles. Lorsque la base de données ne contient que la vidéo 3.4, chaque image est traitée en moyenne en 0.11s. Avec une base de données de plus de 220 vidéos pour un total de plus de 20h de vidéos, les images sont traitées en moyenne en 0.13s. La différence est minime et nous pouvons donc penser que notre programme fonctionnerait avec une énorme base de données.

Perspectives d'évolution

Ce projet étant assez complexe et ambitieux nous avons travaillé sur les fonctions principales de détection, nous avons cependant pensé à plusieurs améliorations ou changements possibles au fur et à mesure que nous avancions. Voici une liste non-exhaustive des idées que nous avons eu :

- Ajouter de nouvelles altérations d'images (format 4/3, image plus applatie ou moins large etc...).
- Diviser les hash de la base de données en plus que des quarts (améliorerait la detection mais prendrait plus de temps).
- Augmenter le nombre d'images par seconde lors de la fragmentation des vidéos pour la BDD (prendrait plus de place et nécessiterait une machine ou un serveur relativement puissant).
- Ajouter un son lorsque la pop-up s'ouvre afin d'alerter l'utilisateur.
- Compacter nos fichiers en un vrai logiciel comprenant un outil de position de curseur facilitant la capture.
- Entrer en collaboration avec les sites de streaming qui peuvent probablement récupérer les images directement sans utiliser de captures.

Points de vue personnels

Sur un plan nous concernant plus directement, nous sommes assez satisfaits de ce que nous avons réussi à produire autour de ce sujet. En effet, le thème nous intéressait beaucoup mais nous ne savions pas trop comment nous y prendre et étions un peu perdu au début du projet. Les recherches que nous avons effectué nous ont permis de débloquer la situation et d'avancer petit à petit vers l'objectif final. Ainsi nous nous sommes rendus compte à quel point il est important de se documenter et réfléchir posément au préalable aux tâches prioritaires. Nous avons également progressé en terme de travail d'équipe, en effet, travaillant sur un programme il nous fallait coder séparément des fonctions qui seraient compatibles entre elles et s'adapter constamment au travail de son binôme. Cela montre à quel point il peut être compliqué de travailler au sein d'un groupe plus grand sans supervision et qu'il est nécessaire que quelqu'un de confiance attribue les tâches et dirige le groupe pour que tout le monde avance dans le même sens. Enfin nous avons également développé nos connaissances et notre intêret pour l'informatique ce qui nous conforte tous deux dans notre choix de spécialité pour notre cycle d'ingénieur.

Bibliographie

[1] https ://pypi.python.org/pypi/ImageHash : Site de la librairie Python ImageHash

[2] https ://pillow.readthedocs.io/en/5.1.x : Site de la librairie Python Pillow

[3] https ://www.ffmpeg.org/ : Site de Ffmpeg

Annexe A

Liens utiles

Voici une petite liste d'url intéressantes au sujet de ce projet :

— www.polytech.univ-tours.fr : Site de Polytech Tours.
— https://www.univ-tours.fr/ : Site de l'université de Tours.
— https://github.com/JohannesBuchner/imagehash : Page Github de la librairie ImageHash.
— http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html : Article comparant les méthodes de "Average Hash" et de "Perception Hash".
— http://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html : Article sur la méthode de "Difference Hash".
— https://fullstackml.com/wavelet-image-hash-in-python-3504fdd282b5 : Article sur une nouvelle méthode, le "Wavelet Hash".
— https://realpython.com/fingerprinting-images-for-near-duplicate-detection/#where-can-image-fingerprinting-be-used : Article sur l'utilisation des hash dans la détection de copie d'images.
— https://github.com/realpython/image-fingerprinting : Page Github d'un projet utilisant ImageHash dans la détection de duplicata d'images.

Annexe B

Fiche de suivi de projet PeiP

Séance n° 1 du 19/01/2018	Rencontre avec l'encadrant ; premières recherche sur la détection de duplicata ; lecture des fichiers joints à l'intitulé du projet ; téléchargement de vidéos.

Séance n° 2 du 26/01/2018	Téléchargement d'une base de données de vidéos ; prise en main de la base de données.

Séance n° 3 du 02/02/2018	Recherche de solutions pour comparer des images ; découverte des hash functions ; premiers essais avec le hash sur Python ; découverte des bases de données shelve avec Python.

Séance n° 4 du 09/02/2018	Création du fichier qui ajoute des hash à notre base de données shelve ; prise en main de cette base de données avec des tests sur différentes images.

Séance n° 5 du 16/02/2018	Mise en place de l'altération d'image avec différentes méthodes comme le niveau de gris, les couleurs inversées ou un effet miroir ; Création du fichier qui détecte si une image est dans la base de données.

Séance n° 6 du 23/02/2018	Mise en place de la méthode des 4 bases de données pour détecter rapidement les images ressemblantes ; modification de fichier de création de base de données et de détection.

Séance n° 7 du 09/03/2018	Prise en main de l'outil ffmpeg pour segmenter les vidéos ; transformation des fichiers pour prendre en entrée une vidéo qui est alors segmentée en images.

Séance n° 8 du 16/03/2018	Modifications de notre programme pour le rendre plus performant ; tests de détection de duplicatas de vidéos.

Séance n° 9 du 23/03/2018	Recherche de moyens pour pouvoir détecter une vidéo en direct ; début de la modification de notre programme dans ce sens.

Séance n° 10 du 30/03/2018	Mise en place de notre outil de capture en direct ; mise en place du critère d'arrêt avec le choix de capturer une image par seconde et d'arrêter au bout de 30 images détectées comme duplicata.

Séance n° 11 du 06/04/2018	Différents tests de notre programme avec des vidéos parfois modifiées comme avec les couleurs changées, des bandes sur les côtés ou une webcam cachant un coin.

Séance n° 12 du 13/04/2018	Autres tests avec différentes tailles de base de données ; Tests de performance.

Algorithme de détection de duplicata sur des vidéos en flux live

Rapport de projet S4

Résumé : La détection de duplicata vidéo en direct consiste en la comparaison d'une suite d'images diffusées sur un stream avec celles d'une base de données créée à cet effet préalablement. Cela peut permettre d'éviter la diffusion de contenu soumis à un copyright par exemple.

Mots clé : duplicata, stream, base de données, copyright

Abstract : Video duplicates detection consists of comparing a sequence of pictures sampled from a stream with those contained in a database generated beforehand for this purpose. This could prevent from the broadcasting of copyrighted content for example.

Keywords : duplicates, stream, database, copyrighted

Auteur(s)

Léo Lefaucheux
[leo.lefaucheux@etu.univ-tours.fr]
Thomas Fournier
[thomas.fournier@etu.univ-tours.fr]

Encadrant(s)

Mathieu Delalandre
[mathieu.delalandre@univ-tours.fr]

Polytech Tours
Département Informatique

Ce document a été formaté selon le format EPUProjetPeiP.cls (N. Monmarché)

École Polytechnique de l'Université de Tours
64 Avenue Jean Portalis, 37200 Tours, France
http://www.polytech.univ-tours.fr