Analyse comparative des différentes versions ULBMP

Arévalo Tania (000570504)

6 May 2024

Contents

1 Introduction

Lors de ce deuxième quadrimestre, nous avons eu l'occasion de travailler sur un projet visant à compresser diverses images de tailles différentes. Pour cela, nous avons travaillé sur quatre formats de décodage et d'encodage, qui modifient les données d'une image, la façon dont les pixels sont représentés, ainsi que d'éventuels changements dans le 'header'.

Le premier format fonctionne d'une manière plutôt simple, c'est-à-dire que les pixels sont traités individuellement. Chacun est représenté par trois octets dans le fichier de l'image. Ces octets définissent l'intensité du rouge, du vert et du bleu (RGB), créant ainsi la couleur du pixel.

Le deuxième est un format de compression en RLE, c'est-à-dire que le nombre de pixels de même couleur se suivant est compté et ajouté dans l'encodage sous forme d'un octet. Nous avons donc quatre octets par groupe de pixels identiques à encoder: ce nombre de répétitions et chaque niveau de saturation pour le rouge, le vert et le bleu.

La troisième version consiste à utiliser une palette qui est ajoutée dans le 'header'. Celle-ci contient les différents pixels de couleurs, dont chacun est défini par les trois octets RGB. Les pixels sont ensuite encodés selon la profondeur donnée. Si celle-ci est inférieure à 24, chaque octet contient entre 1 à 8 pixels différents, les bits formant un pixel étant égaux à l'indice du pixel en question dans la palette. Sinon, il n'y a pas de palette et son encodage est fait en RLE. De plus, la profondeur 8 a à la fois une palette et des octets encodés en RLE.

La quatrième version se concentre davantage sur les voisins de pixels, compare si leurs couleurs sont distinctes ou similaires et les classe dans différents ensembles d'octets en fonction de ces différences. Cette technique peut s'avérer plus utile dans des cas d'images de grande taille avec des pixels différents mais de couleurs proches.

Le but de ce rapport est de comparer les différentes versions entre elles et de discuter (point 4) des résultats (point 3) grâce à différentes méthodes (point 2). Un autre sujet important que nous aborderons est les éventuels problèmes rencontrés lors de la création du projet. Pour comprendre pleinement ce rapport, il est nécessaire d'avoir des connaissances en Python, ses bibliothèques, le fonctionnement de l'encodage et du décodage d'une image, ainsi que des modes d'encodage différents, tel le RLE.

2 Méthodes

Comme mentionné dans l'introduction, nous avons décidé d'utiliser différentes méthodes pour comparer ces versions sur deux images différentes. Les deux images principales que nous avons choisies sont "checkers" et "jellybeans". Ce choix a été fait car elles présentent des différences concernant leur taille (hauteur et longueur), nombre de pixels, ainsi que la quantité de pixels de couleurs différentes.

Ces approches sont les graphiques, les images et les tableaux qui classeront les résultats de nos recherches. Elles ont été choisies car les éléments principaux que nous avons comparés sont les taux de compression, l'espace qu'une image compressée occupe, l'utilité et l'inutilité qu'une version peut avoir pour différentes images, l'encodage des pixels, etc. Elles apporteront de la lisibilité, clarté et faciliteront la compréhension et l'observation des comparaisons.

La première approche que nous avons utilisée pour effectuer ces comparaisons est l'utilisation des graphiques de la bibliothèque Python Matplotlib. Grâce à ces derniers, nous avons la possibilité d'observer plus clairement les résultats recherchés et d'en discuter. Pour les créer, nous avons utilisé le squelette de l'algorithme suivant:

main.py

```python
# squelette de l'algorithme principal pour créer des graphiques

import matplotlib.pyplot as plt

plt.title("Titre") # titre du graphique
plt.plot([1,2,3,4], [1, 2, 3, 4]) # listes contiennent les coordonnées de longueur et hauteur du graphique
# longueur: versions
# hauteur: taille des fichiers images compressés ou le temps que cela prend pour compresser de la version 1 en les autres versions
plt.ylabel('') # titre de l'axe y
plt.xlabel('') # titre de l'axe x
plt.show()
```

Les graphes sont sauvegardés et ajoutés sous forme d'image.

Tout comme les graphiques, les tableaux servent à structurer et regrouper tous les résultats trouvés pour que nous puissions les comparer entre eux. La différence est que les tableaux s'avèrent utiles lorsque nous voulons comparer des versions textuellement et ils permettent d'avoir une classification des résultats plus librement. Les tableaux sont complétés avec des nombres que nous pouvons trouver en regardant les informations des fichiers (taille, nombre total de bytes), mais aussi en utilisant le 'hexdump -C' dans le terminal pour voir le contenu des fichiers images et pouvoir séparer header et pixels.

Ensuite, une autre méthode qui nous a aidés est l'utilisation d'images, car elles offrent la possibilité d'avoir des illustrations des résultats finaux du programme. Grâce à elles, nous pourrons créer des graphiques et des tableaux. Les graphiques et les codes seront sauvegardés au format .png pour pouvoir être ajoutés dans le rapport.

Pour finir, pour calculer le temps que l'exécution du code prend, nous avons ajouté dans window.py du projet 2, le module time ('import time') et en trois petites lignes nous avons calculé le temps pour chaque version. Voici la ligne de code ajoutée: print("time", time.time() - start), avec start (ligne de code ajoutée une fois que le nom de fichier dans lequel nous voulons ajouter l'image compressée est donné) : start = time.time().

3 Résultats

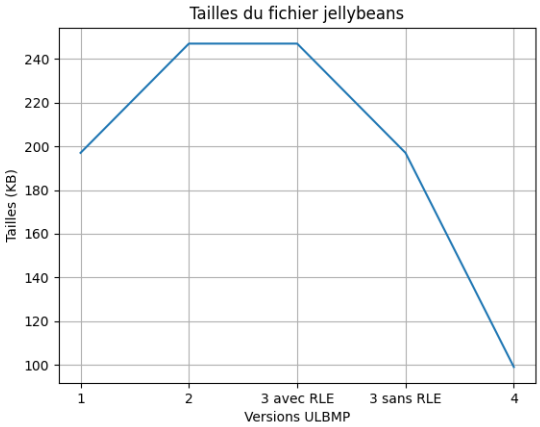
Les tableaux suivants présentent les taux de compression, le nombre de bytes nécessaires pour encoder les pixels (sans compter la palette s'il y en a une), ainsi que la taille du fichier dans les différentes versions de compression.

version	taux de compression	total bytes pour pixels	taille
ULBMP 1.0	(nombre total de pixels) * 3	921600 bytes	922 KB
ULBMP 2.0	taux de compression +/- 12 (comparé à 1.0)	+/- 76800 bytes	77 KB
ULBMP 3.0, sans RLE	taux de compression +/- 24 (comparé à 1.0)	+/- 38400 bytes	38 KB
ULBMP 4.0	taux de compressions +/- 2,5 (comparé à 1.0)	+/- 364709 bytes	365 KB

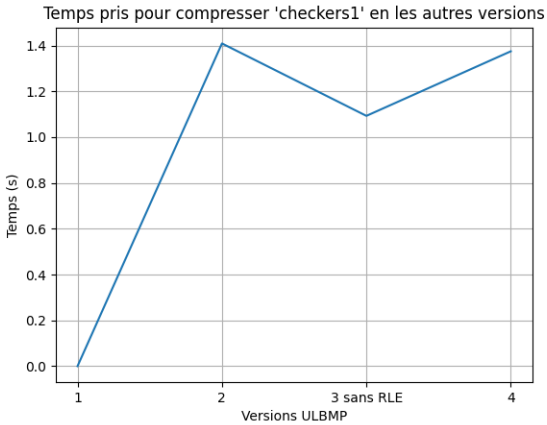
TABLEAU 1 - compare taux de compression, les bytes encodant les pixels et la taille des fichiers "checkers"

version	taux de compression	total bytes pour pixels	taille
ULBMP 1.0	(nombre total de pixels) * 3	+/- 196608 bytes	197 KB
ULBMP 2.0	taux de compression +/- 0,8 (comparé à 1.0)	+/- 246768 bytes	247 KB
ULBMP 3.0, sans RLE	taux de compression +/- 1 (comparé à 1.0)	+/- 196608 bytes	197 KB
ULBMP 3.0, avec RLE	taux de compression +/- 0,8 (comparé à 1.0)	+/- 246768 bytes bytes	247 KB
ULBMP 4.0	taux de compressions +/- 1,9 (comparé à 1.0)	+/- 99144 bytes bytes	99 KB

TABLEAU 2 - compare taux de compression, les bytes encodant les pixels et la taille des fichiers "jellybeans"



GRAPHIQUE 1 - compare tailles des fichiers "jellybeans", créé à l'aide du squelette partie 2.1



GRAPHIQUE 2 - compare le temps pris pour compresser "checkers1" en 2.0, 3.0 sans RLE, 4.0 (toutes les compressions sont faites à partir de 1.0)

4 Discussion

Dans cette section, nous analysons les résultats trouvés dans la partie précédente, les hypothèses que nous pouvons en déduire, et comment nous sommes à ces chiffres.

Pour commencer, nous savons que la compression d'une image est la réduction de sa taille, ceci peut être fait de différentes manières, mais ce que les tableaux et graphiques nous montrent, c'est qu'il y a un rapport avec la diminution du nombre de bytes qui représentent les pixels. Chaque version a une approche différente à cet encodage, mais ce n'est pas pour autant que plus l'une d'entre elles est complexe, plus efficace elle sera. Pour prouver ceci, nous nous sommes concentrés sur les résultats de ces deux images et nous avons vu elles n'ont pas les mêmes résultats.

Dans le cas de "checkers", nous pouvons voir qu'avec la version 3, nous avons un taux de compression +/- 24 fois plus petit comparé à la version 1 (encodant chaque pixel avec trois bytes).

En effet, si nous choisissons une profondeur correspondante à la création d'une palette de taille parfaite (dans ce cas la profondeur serait de 1) et disons que l'encodage doit être fait sans RLE, le résultat sera meilleur. Ceci est le cas, parce que nous avons seulement deux types de pixels, la palette est petite, donc ils peuvent être encodés plus facilement et ne prennent plus autant de place; un byte représenterait 8 pixels au lieu d'un tiers d'un seul pixel.

C'est en faisant la différence des nombres totaux de bytes les représentant que nous avons un le résultat du taux de compression. Bien que pour les autres versions la compression était aussi importante, la version 3 (de +/- 38400 bytes de pixels) transforme l'image de 922 KB (1.0) en une de 38 KB (avec +/- 38400 bytes de pixels). Ce nombre est plus significatif que ceux donnés par les versions 2 et 4, respectivement 77 KB (+/- 76800 bytes de pixels) et 365 KB (+/- 364709 bytes de pixels).

Bien que la version 2 utilise la RLE, ce qui pourrait être intéressant dans "checkers", puisqu'il y a que des groupes de pixels identiques, quatre bytes contiennent majoritairement que 16 pixels, ce qui est 50 pour-cent de ce que 4 bytes de 3.0 peuvent contenir.

La raison pour laquelle l'encodage de la version 4 n'est pas très nécessaire pour cette image est parce que les deux couleurs sont noire est blanche (totalement opposées) et ces pixels se suivent ou pas. Le code est programmé pour les encoder de deux façons, soit comme un "nouveau pixel" (quand ils ne sont pas identiques, mais se suivent) en 4 bytes, soit comme un "pixel de très grande différence" en 3 bytes. Le résultat de ceci donnera une image de taille 365 KB.

Par contre, dans le cas de "jellybeans", le tableau 2 et le graphique 1 nous montrent que la meilleure compression est obtenue par la quatrième version. Celle-ci transforme "jellybeans1" de 197 KB (+/- 196608 bytes de pixels) en 99 KB (99144 bytes de pixels), comme nous le savons dans la version 1 chaque pixel est encodé en trois bytes, mais dans la version 4, l'encodage est fait comme avec "checkers4", mais nous avons plus de façons de faire ceci, pas seulement "nouveau pixel" (4 bpp) ou les trois types de "pixel de très grande différence" (3 bpp). Il y a aussi l'encodage de "petite différence" (1 bpp) et de "différence intermédiaire" (2 bpp). La raison pour laquelle cette image peut donc passer à un taux de compression 1,9 fois plus petit est parce que le plus souvent l'encodage sera fait en moins de 3 octets, nous pouvons affirmer cela en regardant la colonne 'total bytes pour pixels'.

Une autre observation intéressante et différente à l'analyse dans le premier tableau est que parfois ce sont les versions les plus simples (1.0) qui font un travail plus efficace. Nous pouvons voir que lorsque nous voulons inclure du RLE dans l'encodage des pixels, le nombre total d'octets de pixels peut s'avérer plus important que ceux dans 1.0. Ceci arrive quand nous avons une image sans pixels identiques qui se suivent (pire des cas) ou quand il y en a très peu. Dans ce cas-ci, ce type de version aurait comme effet l'augmentation de la taille du fichier. Et comme l'indique le tableau 2, nous remarquons une différence avec 1.0 ayant une taille de 197 KB et 2.0, 3.0 avec RLE ayant une taille de 247 KB. Ces deux dernières ont les pixels encodés en 4 bytes, ce qui crée un taux de compression 33 pour-cent plus grand que celui de 1.0 et 3.0 sans RLE.

Ces résultats prouvent donc notre hypothèse qui dit que ces versions différentes apporteront différents résultats selon le fichier donné. Chaque image, selon ses propres caractéristiques, peut être très bien compressée grâce à une version comparé aux autres, mais pour une autre image ça ne sera peut-être pas le cas, comme nous pouvons le voir avec "checkers" et "jellybeans". Parfois, il faut aussi se tourner vers un format plus basique.

Une autre comparaison à mentionner concerne le temps nécessaire à chaque version pour compresser et sauvegarder une image, initialement en une version A. Nous constatons que le temps utilisé diffère toujours, mais il reste assez court.

Le graphique 2, contient les résultats pour l'image "checkers". La version 1.0 est initialisé au temps 0, car c'est cette version originale qui sera être transformée en chaque autre version.

Pour la version 2.0, nous constatons que cela prend environ 1.41 seconde, ce qui est la plus grande quantité de temps, comparé aux autres formats. La raison en est qu'il faut compter les pixels identiques qui se répètent avant d'encoder et de sauvegarder.

Pour la version 3 sans RLE, le temps mis est le plus bas, c'est environ 1.09 seconde, car à part la création de la palette, il n'y a pas de comptage RLE à faire (c'est même impossible d'utiliser RLE car il n'y a que deux types de pixels). Une autre chose qui aide est le fait que 'checkers' est constitué de deux variantes de pixels se répétant, ce qui ne rend pas la palette très grande.

Enfin, dans la dernière version, le temps nécessaire est de 1.37 seconde, ce qui est très proche de la deuxième. Dans ce cas, chaque pixel sera encodé comme mentionné précédemment, ce qui prend en effet une quantité de temps plus importante.

En analysant ce résultats, nous observons bien la différence de taille du code, surtout pendant son exécution. Et cette différence peut être remarquée dans la quantité de temps nécessaire (graphique), bien que ces quantités ne soient pas alarmantes, elles montrent que certaines versions sont bien plus complexes que d'autres et nécessitent un travail plus important.

En comparant le graphique et le tableau décrivant 'checkers', nous pouvons remarquer que la version 3.0 sans RLE est la meilleure compression possible pour ce fichier, contrairement aux autres. Cela est dû au fait que cette compression est plus rapide que les autres et la taille en KB est beaucoup plus petite que pour les autres versions. Cependant, cela peut varier d'une image à l'autre.

5 Conclusion

Pour conclure ce rapport, il est notable que même si chaque version a le même but final (compresser une image), elles présentent toutes des différences et des similitudes qui requièrent une analyse approfondie de graphiques et de tableaux de comparaisons. Grâce à des modules Python tels que 'matplotlib', ces graphiques peuvent être réalisés en toute facilité. Ces méthodes sont très efficaces lorsque l'on veut illustrer des comparaisons, offrant plus de lisibilité, propreté et compréhensibilité. Un autre module de python que nous avons intégré est le module time, qui permet de calculer le temps d'exécution du code.

Les tableaux de taux de compression ont révélé que, même si un format de compression est le optimal pour une image, ce n'est pas toujours le cas pour une autre. Ceci arrive non seulement parce que deux images peuvent être totalement différentes, mais aussi car l'effet d'une version sur ces deux images agit différement. Pour choisir une version de compression correcte, il est nécessaire d'observer les différentes caractéristiques d'une image. Par exemple, pour une image comportant de nombreux groupes de pixels identiques, un format avec une RLE serait plus intéressant plutôt que d'un autre,.

C'est ces différents facteurs mettent en évidence à quel point les versions diffèrent l'une de l'autre et pour prouver cela, les images choisies avaient étés 'checkers' et 'jellybeans' car elles étaient très différentes l'une de l'autre. Pour 'checkers', la version qui l'encodait le mieux était aussi la version qui prenait le moins de temps à exécuter ce travail, c'était la version 3.0 sans RLE. En revanche, pour 'jellybeans', la version 4 compressait le mieux.

Malheureusement, nous avons eu une légère difficulté dans le code du projet 2, ce qui a impacté le travail sur notre rapport. Nous avions omis d'ajouter lors de la sauvegarde en fichier d'images compressées (window.py), la version 4. Donc même si le code était bon, la sauvegarde en 4.0 ne pouvait pas être faite. Puisque ce n'était qu'un rajout d'un tiers de ligne (elif version == 1 or version == 2 or version == 4: ... au lieu de elif version == 1 or version == 2:...), nous avons corrigé cette erreur afin de pouvoir calculer le temps d'exécution pour 'checkers'.