Projet d'Algorithmique avancée 2016
Compression RLE

Nehari Mohamed

20 décembre 2016

Table des matières

1 Introduction au projet RLE

1.1 Algorithme RLE

La méthode de compression RLE (Run Length Encoding, parfois notée RLC pour Run Length Coding) est utilisée par de nombreux formats d'images (BMP, PCX, TIFF). Elle est basée sur la répétition d'éléments consécutifs.
Le principe de base consiste à coder un premier élément donnant le nombre de répétitions d'une valeur puis le compléter par la valeur à répéter (Définition Internet).
En comparaison à l'algorithme classique qui code le nombre d'occurrence de chaque octet quitte parfois à doubler la taille du fichier de départ dans le pire des cas, j'ai choisi d'appliquer la méthode vu en cour qui consiste à coder les répétition uniquement si elle sont supérieurs à 2 occurrences du même octet, et coder les suites d'octet qui diffèrent en insérant le nombre de caractères qui diffère en négatif, puis les caractères en eux mêmes.

1.2 Description du projet

Le projet consiste à implémenter l'algorithme RLE sur les images. Je dois mettre en place la compression et la décompression des images en fonction de deux mode RGB ET HSV. En comparaison à l'algorithme classique qui analyse les suite d'octet en prenant comme référence le pixel, je devais parcourir les données sur trois plans différents, R puis G puis B et de la même manière pour une image en HSV. Une fois l'algorithme implémenté, je dois comparer les résultats obtenu avec les résultats d'autres algorithmes de compressions bien connu tel que “LZW” et “Huffman”.

2 Analyse du programme

Afin de réaliser ce projet, je me suis aidé du code de Mr Bourdin que j'ai récupéré sur la page de programmation impérative 2 2015 afin de m'en servir comme base pour mon programme.

2.1 Fonction ImageLoad_PPM

La fonction ImageLoad_PPM a été créé par Mr Bourdin, elle permet de charger les données d'une image PPM. Plusieurs vérifications sont effectuées au préalable afin de vérifier si l'image est bien au format PPM. Pour la réalisation de mon programme, j'y ai ajouté le stockage des données du Header de l'image et le calcul de la taille de celui-ci étant données que cette partie est indépendante des pixels de l'image.

2.2 Fonction sizeOfHeader

La fonction SizeOfHeader parcours le Header de l'image afin de connaître le nombre d'octet qui le compose. Le résultat rendu sera stocké dans le champ "int sizeHeader" de la structure "Image".

2.3 Fonction compress

La fonction compress est une des fonction principale du programme, elle permet la compression RLE des données de l'images qui ont été chargées.
La première étape consiste à écrire le header de l'image dans le fichiers compressé, étant donné que le processus de compressions des pixels ne s'applique pas à lui.
Une fois cette étape effectuée, on entre dans la boucle de compression des pixels.
On se déplace de trois en trois afin de comparer les valeurs ciblé de chaque pixels, R avec R, G avec G, et B avec B, et ce pareil pour HSV.

2.3.1 Répétition

Dans la boucle principale, une boucle while se charge de parcourir les données tant qu'une répétition est trouvée.
A chaque répétition, on incrémente la variable "répétition" jusqu'à

ce que les répétitions s'arrêtent ou que l'on ai atteint un maximum de 127 répétitions.
On compare ensuite le nombre d'occurrences trouvée afin de voir si le nombre est supérieur au seuil minimal requis. Si le nombre de répétition est supérieur ou égal au seuil, on écrit le nombre de répétition suivi du caractère dans le fichier compressé.

2.3.2 Non Répétition

Une autre boucle while s'occupe de parcourir les données afin de compter le nombre de non répétition.
Si les caractères analysés ne se répètent pas, la variable "noRepetition" est incrémentée jusqu'à trouver au moins deux répétitions successives, ou jusqu'a trouver un maximum de non répétition de 128.
Une fois sortie de la boucle de non répétition, on regarde si le nombre de non répétition est supérieur à 0.
Si oui, on en écrit ce nombre en négatif suivi des caractères non répétés dans le fichier compressé.

2.3.3 Fin du parcours d'un champ

Une fois que toutes les données d'un champs ont été parcourues, on insère le code de contrôle "0" afin de afin de prévenir le décompresseur du changement de champ et donc qu'il faudra qu'il revienne au début des données et passés au champ suivant, puis on revient au début en passant a l'analyse du champ suivant

2.4 Fonction decompress

La fonction decompress permet de décompresser un fichier qui à préalablement été compressé via la fonction compress.
La première étape consiste en la création de trois fichier temporaires "tempR", "tempG", et "tempB" destinés à accueillir pour chacun son champ respectif.
Au départ, j'avais décidé de décompresser chaque champ en me déplaçant au fur et à mesure dans le fichier compressé afin de placer chaque champs de chaque pixel à sa place respective via fseek(), cependant avec ce fonctionnement le temps d'exécution était beaucoup trop élevé, et j'ai donc décidé de privilégié le temps d'exécution à l'espace utilisé.

2.4.1 Decompression du fichier compressé

La boucle principale lit le fichier compressé jusqu'à la fin de celui-ci.
A chaque code de répétition lu, on teste si le nombre est positif ou négatif, s'il est positif, alors on écrit l'octet suivant autant de fois qu'il y a de répétitions, s'il est négatif, on continue la lecture en écrivant l'octet lu autant de fois qu'il y a de non répétition.
Si le code de contrôle "0" est lu, cela indique au décompresseur qu'il doit passer à la décompression du champ suivant, et les résultats en sortie seront donc écrits dans leurs fichiers temporaires respectifs.

2.4.2 Regroupement des trois champs

Une fois la décompressions terminée, on regroupe les trois fichiers temporaires en un seul fichier en prenant un champ de chaque fichier à tour de rôle, afin de reconstituer un pixel à chaque tour de boucle.
Enfin, une fois le fichier original reconstitué, les fichiers temporaires n'étant plus utiles sont supprimés.

2.5 Fonctions RGB_TO_HSV et HSV_TO_RGB

la fonction RGB_TO_HSV parcoure les données de l'image, et convertit chaque pixel RGB en HSV, et réciproquement pour la fonction HSV_TO_RGB.
Ne connaissant rien sur le mode HSV, j'ai effectué plusieurs recherche afin de comprendre comment passer de RGB en HSV, ce qui m'a mené à trouver pas mal de formules mathématiques et procédé afin d'effectuer cette conversion.
Cependant ayant un niveau médiocre en mathématique, je n'ai pas réussi à implémenter ces fonctions.
Par conséquent, j'ai dû prendre ces deux fonctions sur internet, que j'ai adapté à mon programme. Ainsi aucun de ces procédés de conversion ne sont de moi.

Les liens des fonctions sont disponibles dans la parties « Sources ».

3 Comparaison des résultats avec d'autres algorithmes de compression sans pertes

3.1 Algorithme LZW

3.1.1 RLE vs LZW en mode RGB

	Original Size	RLE	LZW 9 bits	LZW 10 bits	LZW 11 bits	LZW 12 bits
Chatou.ppm	2 359 312	2 506 103	2 267 759	2 303 307	2 327 768	2 269 995
Frame01.ppm	196 623	3 699	5 632	2 740	2 048	2048
Garden.ppm	540 015	538 643	499 845	513 285	511 365	499 504
Kili.ppm	2 359 312	2 442 372	2 036 993	2 058 425	2 080 004	2 014 741
Pics.ppm	205 944	28 880	14 573	10 552	8 954	8 284
Refuges.ppm	2 359 312	2 365 697	1 940 777	1 896 887	1 863 329	1 784 692
Requin.ppm	2 359 312	2 478 609	2 419 748	2 506 194	2 534 360	2 534 606
TeaPot.ppm	786 447	243 932	330 409	321 520	272 559	247 553
Tulipe.ppm	562 515	495 251	486 533	496 549	489 501	479 967
fruits.ppm	1 396 680	1 226 510	1 040 032	1 069 540	1 084 513	1 047 922
monochrome.ppm	372 075	380 568	364 041	380 779	384 816	384 959
tableau.ppm	799 719	836 224	448 927	398 177	372 445	341 429

En analysant les résultats des deux algorithmes sur des compressions d'images en mode RGB, je peux facilement déduire que l'algorithme LZW reste plus performant que l'algorithme RLE sur ce cas de figure.
Parmi toutes les images, une seule se démarque en compression RLE et elle comporte beaucoup de répétitions.
Trois images ont une compression assez proche de la compression LZW et ont donc des résultats assez homogènes.
Enfin, sur les dix images, huit d'entres elles ont une compressions moins efficace en RLE qu'en LZW malgré la présence d'un nombre important de répétition sur certaines d'entre elles.

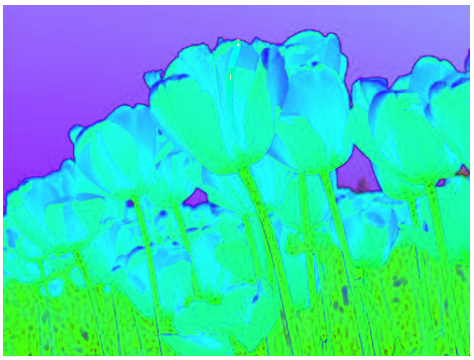
3.1.2 RLE vs LZW en mode HSV

	Original Size	RLE	LZW 9 bits	LZW 10 bits	LZW 11 bits	LZW 12 bits
Chatou.ppm	2 359 312	2 489 165	2 349 430	2 439 331	2 503 054	2 510 875
Frame01.ppm	196 623	4 487	7 236	3 624	2 925	2 534
Garden.ppm	540 015	467 356	472 893	482 252	475 966	466 928
Kili.ppm	2 359 312	2 469 065	2 067 352	2 087 607	2 111 968	2 055 901
Pics.ppm	205 944	16 146	18 147	12 316	9 754	8 905
Refuges.ppm	2 359 312	2 314 040	1 922 927	1 905 000	1 886 129	1 822 436
Requin.ppm	2 359 312	2 465 412	2 401 266	2 481 485	2 526 288	2 528 376
TeaPot.ppm	786 447	251 429	328 292	317 011	266 280	241 104
Tulipe.ppm	562 515	457 891	455 977	456 242	441 776	427 891
fruits.ppm	1 396 680	1 040 371	1 028 844	1 050 509	1 055 215	1 005 565
monochrome.ppm	372 075	336 648	351 026	364 402	397 459	361 557
tableau.ppm	799 719	546 635	413 348	361 694	328 204	300 721

En analysant les résultats des deux algorithmes sur des compressions d'images en mode HSV, je peux déduire que les résultats sont moins alarmant que pour la compression en mode RGB.
Sur les 10 images, 2 d'entre elles ont une compression plus efficace en RLE qu'en LZW, six d'entre elles ont une compression proche de la compression LZW, et seulement quatre ont une compression moins efficace en RLE qu'en LZW.
L'augmentation des répétitions dû à la conversion HSV à améliorer les résultats de l'algorithme RLE, cependant LZW garde malgré tout une avance sur RLE.

Voici l'exemple d'une conversion RGB en HSV ou les répétitions sont augmentées :

(a) Image RGB



(b) Image HSV

3.1.3 Conclusion RLE vs LZW

En conclusion, il est évident que l'algorithme RLE est bien plus efficace lorsqu'il y à de nombreuses répétitions dans l'image.
La conversion d'une image RGB en mode HSV accentue les répétitions et rend les compressions plus fructueuses, ce qui explique cette différence d'écart entre la compression des deux modes.

Cependant, l'algorithme LZW reste tout autant efficace pour les images comportant de nombreuses répétitions que pour des images avec des répétitions moins fréquentes comparé à l'algorithme RLE qui n'est avantageux que si il y a un certaine quantité de répétitions dans l'image.
Le dictionnaire pouvant accueillir un très large choix de séquence, il n'est pas rare de retomber sur une séquence déjà enregistré même si l'image ne comporte pas beaucoup de répétitions, et c'est ce qui explique l'avantage de LZW sur les images RGB n'ayant pas beaucoup de répétitions.
De plus, l'écriture des codes se fait bit par bit, avec la possibilité de choisir le nombre de bit maximum sur lesquels les codes seront écrit, et c'est la un grand gain en espace que procure l'algorithme LZW.

3.2 Algorithme Huffman

3.2.1 RLE vs Huffman en mode RGB

	Original Size	RLE	Huffman
Chatou.ppm	2 359 312	2 506 103	1 957 697
Frame01.ppm	196 623	3 699	26 637
Garden.ppm	540 015	538 643	537 809
Kili.ppm	2 359 312	2 442 372	2 241 443
Pics.ppm	205 944	28 880	43 051
Refuges.ppm	2 359 312	2 365 697	2 194 050
Requin.ppm	2 359 312	2 478 609	2 310 117
TeaPot.ppm	786 447	243 932	610 012
Tulipe.ppm	562 515	495 251	520 062
fruits.ppm	1 396 680	1 226 510	1 274 981
monochrome.ppm	372 075	380 568	363 594
tableau.ppm	799 719	836 224	531 962

En observant les résultats entre l'algorithme Huffman et RLE je peux observer que l'algorithme RLE est la encore plus compétent que Huffman pour les images comportant de nombreuses répétitions. Cet écart de résultats à l'air d'être un peu plus conséquent que pour l'algorithme LZW. Sur les dix images, quatre ont une compression RLE moins efficace qu'avec l'algorithme Huffman, et ces quatre images ont toutes peut de répétitions.

3.2.2 RLE vs Huffman en mode HSV

	Original Size	RLE	Huffman
Chatou.ppm	2 359 312	2 489 165	2 193 897
Frame01.ppm	196 623	4 487	33 767
Garden.ppm	540 015	467 356	525 574
Kili.ppm	2 359 312	2 469 065	2 188 355
Pics.ppm	205 944	16 146	41 179
Refuges.ppm	2 359 312	2 314 040	2 156 228
Requin.ppm	2 359 312	2 465 412	2 244 080
TeaPot.ppm	786 447	251 429	560 325
Tulipe.ppm	562 515	457 891	419 903
fruits.ppm	1 396 680	1 040 371	1 312 419
monochrome.ppm	372 075	336 648	342 846
tableau.ppm	799 719	546 635	449 495

Les résultats de la compression RLE sur les images à fortes répétition sont toujours prédominant.
En convertissant les images en HSV, l'écart de la taille entre les images possédant peu de répétitions en compression RLE et Huffman en RGB c'est aminci en HSV.

3.2.3 Conclusion RLE vs Huffman

Au vu des résultats, je peux comme pour LZW affirmer que l'algorithme RLE est compétent lorsqu'il a beaucoup de répétitions dans le fichier, comparé à l'algorithme Huffman.
Huffman garde un certain avantage sur les images avec peu de répétitions comparé à RLE.

3.3 Algorithme Shannon-Fano

3.3.1 RLE vs Shannon-Fano en mode RGB

	Original Size	RLE	Shannon-Fano
Chatou.ppm	2 359 312	2 506 103	1 971 331
Frame01.ppm	196 623	3 699	26 613
Garden.ppm	540 015	538 643	539 319
Kili.ppm	2 359 312	2 442 372	2 252 162
Pics.ppm	205 944	28 880	43 330
Refuges.ppm	2 359 312	2 365 697	2 205 206
Requin.ppm	2 359 312	2 478 609	2 322 025
TeaPot.ppm	786 447	243 932	611 818
Tulipe.ppm	562 515	495 251	523 795
fruits.ppm	1 396 680	1 226 510	1 285 174
monochrome.ppm	372 075	380 568	364 628
tableau.ppm	799 719	836 224	533 443

3.3.2 RLE vs Shannon-Fano en mode HSV

	Original Size	RLE	Shannon-Fano
Chatou.ppm	2 359 312	2 489 165	2 207 310
Frame01.ppm	196 623	4 487	34 369
Garden.ppm	540 015	467 356	527 996
Kili.ppm	2 359 312	2 469 065	2 207 569
Pics.ppm	205 944	16 146	41 308
Refuges.ppm	2 359 312	2 314 040	2 172 602
Requin.ppm	2 359 312	2 465 412	2 257 298
TeaPot.ppm	786 447	251 429	568 528
Tulipe.ppm	562 515	457 891	480 578
fruits.ppm	1 396 680	1 040 371	1 318 071
monochrome.ppm	372 075	336 648	344 477
tableau.ppm	799 719	546 635	451 434

3.3.3 Conclusion RLE vs Shannon-Fano

La conclusion est exactement identique à celle de l'algorithme de Huffman.
Je penses que cela est compréhensible du fait que Huffman et Shannon-Fano sont très proche.
L'algorithme RLE reste encore compétent lorsqu'il y a beaucoup de répétitions dans le fichier, comparé à l'algorithme Shannon-Fano.
Tout comme Huffman, Shannon-Fano garde également un certain avantage sur les images avec peu de répétitions comparé à RLE.

3.4 Conclusion sur l'efficacité de l'algorithme RLE

Après avoir comparé les résultats de l'algorithme RLE avec les résultats d'autres algorithme sans perte, je peux conclure avec évidence que RLE tire sa force dans la répétition des occurrences.
Plus il y aura de répétitions dans un fichier, plus l'algorithme RLE sera efficace et donnera de meilleur résultats comme on a pu s'en rendre compte avec la conversion de RGB à HSV.
Malheureusement, RLE sera beaucoup moins efficace sur un fichier qui présente peu de répétitions, et c'est ce qui fait sa faiblesse comparé à LZW par exemple qui est plus complexe ou d'autres algorithmes de compression sans perte, qui présentent l'avantage d'être compétent aussi bien pour les fichiers avec beaucoup de répétitions que pour les fichiers qui en possèdent peu.

4 Sources

1. LZW : https://github.com/dbry/lzw-ab
2. Huffman : http://malm.tuxfamily.org/huffman.htm
3. Shannon-Fano : https://github.com/macton/shannon-fano
4. RGB_TO_HSV : https://gist.github.com/fairlight1337/4935ae72bcbcc1ba5c72
5. HSV_TO_RGB : https://gist.github.com/fairlight1337/4935ae72bcbcc1ba5c72