Dodane Jean-Baptiste

TS_3

TPE 2001-2002

Cahier de bord

LES MÉTHODES DE COMPRESSION GRAPHIQUE ET LES FORMATS INFORMATIQUES CONCERNÉS

Professeurs :
M. Thiebert
M. Jussiaux

Horaires :
Jeudi 08h00 - 09h50

Résumé des séances

- 13/09/01 : Première séance : Choix du sujet, premières recherches.
- 20/09/01 : Recherche des éléments et notions indispensables pour comprendre les compressions.
- 27/09/01 : Les différents formats, utilisation, et comparaison.
- 04/10/01 : Les algorithmes LZ**.
- 11/10/01 : Les algorithmes
- 18/10/01 : Recherches diverses.
- 25/10/01 : Le prédictif.
- 01/11/01 : *Pas de séance (Jour férié).*
- 08/11/01 : Commencement d'un programme de compression RLE.
- 15/11/01 : Étude des fichiers.
- 22/11/01 : La compression RLE (Delphi).
- 29/11/01 : *Pas de séance (Olympiades de la Physique).*
- 06/12/01 : *Pas de séance (Forum du Haut-Doubs).*
- 13/12/01 : Programme Delphi : traitement de l'image.
- 20/12/01 : Programme Delphi : problème liés au débordement de pile et à l'enregistrement des fichiers créés.
- 10/01/02 : *Présentation*
- 17/01/02 : La compression LZW (Delphi).
- 24/01/02 : Programme Delphi : traitement de l'image.
- 31/01/02 : Programme Delphi : problèmes et améliorations
- 21/02/02 : Recherche d'applications de la compression graphique dans le domaine des images satellites.

Séance du 13/09/01

08h00 – 09h00 : Salle D19

Avec Cyril Roussillon et Christophe Dumont, nous avons choisi un sujet concernant les applications informatiques, étant tous deux intéressés par cet environnement.

Pour se mettre en relation avec le thème de l'image, nous avons choisi les fichiers graphiques. Nous connaissons les principaux formats qui circulent sur Internet, le JPEG, GIF, BMP, … et quelques autres. Mais plusieurs formats signifient plusieurs utilisations, et nombreuses différences.

La partie intéressante des formats est la compression. Comment peut-on enregistrer une image dans un fichier qui prend moins de place que le fichier original ?

Il y a différents formats, donc différentes méthodes, que nous connaissons légèrement. Nous avons choisi d'étudier les procédés qui permettent aux logiciels de traitement des images de compresser ces données.

09h00 – 09h50 : Salle informatique D09

Après quelques recherches sur Internet, nous avons remarqué deux types de formats différents : les formats utilisant la compression conservative, et les autres utilisant la compression non conservative.

La compression conservative permet, après un cycle de compression/décompression, de retrouver une copie exacte des données initiales.

La compression non conservative autorise des pertes par rapport aux données initiales. Par exemple, une image enregistrée en JPEG montre quelques différences (netteté, précision) suivant le taux de compression choisi. L'intérêt de cette compression est de garder une assez bonne qualité de restitution, tout en réduisant la taille (mieux que la compression conservative) de l'image. Cette compression ne s'applique quasiment qu'aux images. Un fichier texte auquel il manquerait des mots est illisible, un fichier exécutable avec quelques octets en moins est détruit …

Je me tournerais plus vers les compressions conservatives, et Cyril plus vers le format JPEG (compression non conservative).

J'ai commencé à chercher des informations générales sur les principaux formats.

Séance du 20/09/01

08h00 – 09h50 : Salle informatique D09

J'ai commencé par rechercher des caractéristiques générales des formats et les notions indispensables dans le domaine de l'imagerie graphique

Représentation : La représentation de l'image peut être sauvegardée dans 2 structures différentes : le BITMAP, où chaque point de l'image est codé, et le VECTORIEL, où la description seulement est codée. Par exemple, un carré qui sera enregistré en bitmap donnera un fichier contenant tous les points de ce carré, alors qu'un format vectoriel enregistrera seulement 2 points, et peut-être une couleur, une épaisseur de trait …

On voit immédiatement que le second fichier est beaucoup plus souple et léger. De plus, le zoom sur l'image est très efficace : puisqu'on a les coordonnées, on obtient pas le phénomène de "pixellisation" en recalculant les formes. Mais il ne supporte que les schémas simples, et il se révèle être difficile de traiter une partie de l'image, car on ne réfléchit plus en points, mais en formes (qui peuvent plus ou moins grandes, quelconques, …).

Le format vectoriel étant déjà de la compression lors de la création et la modification du fichier, je ne vois pas quoi traiter sur ce sujet. Par contre, le bitmap me paraît (parce qu'il existe pour ça !) particulièrement adapté au traitement par points.

La couleur : On utilise le codage RGB (Red – Green – Blue) (Soit RVB en français). Ce triplet permet de définir une couleur : Chaque couleur peut prendre une valeur de 0 à 255, indiquant son intensité. Par exemple, un triplet (255, 0, 0) représente la couleur rouge foncé. De même, (0, 0, 0) représente le noir et (255, 255, 255) le blanc. C'est ce qu'on appelle la synthèse additive (Rouge + Vert + Blanc) à laquelle notre œil est sensible. Grâce à ce triplet, on peut générer $(2^8)^3$ couleurs, soit parfaitement ce que l'œil humain est capable de voir (d'après le site Web *http://cedric.cnam.fr/~farinone*).

Cette échelle additive s'imagine facilement en ajoutant des couleurs à la couleur noire. Donc plus on ajoute de couleurs (R,V,B), plus on se rapproche du blanc. Elle est utilisée pour les écrans. Elle s'oppose à l'échelle soustractive, où l'on soustrait des couleurs à la couleur blanche (CMYK : Cyan, Magenta, Yellow, blacK ; soit CMJN en français). Cette quadrichromie est le principe utilisé par les imprimantes couleurs, la photographie. On trouve également des autres modèles, comme le YUV (Luminance et Chrominances), utilisé pour les téléviseurs (et le format JPEG), et HSV (ou HSB) : Hue, Saturation, Brightness.

Les systèmes les plus simples paraissent être le RGB et CMYK. Le RGB est utilisé pour les écrans, donc pour les bitmaps.

J'ai trouvé en même temps les notions de compression asymétriques, codages adaptatifs, mais je pense réserver les compressions pour les prochaines séances, après avoir acquis les connaissances élémentaires.

Cyril a commencé les premières fonctions d'un programme utilisant la compression JPEG. De mon coté, je ne sais pas encore si je vais faire quelque chose sous Delphi. J'attends de voir les compétences requises.

Séance du 27/09/01

08h00 – 09h50 : Salle informatique D09

Recherche d'informations sur les différents formats graphiques.

Le BMP :

Caractéristiques : C'est le plus simple des fichiers. Il existe une version pour Windows et une autre pour OS. Il est facilement modifiable, permet de faire des conversions vers tous les autres fichiers, puisqu'il est la base du graphisme informatique. Il est aussi très lourd.

Compression : Il peut utiliser un type de compression : le RLE (Run Length Encoding). Le principe est de lire les pixels, et de supprimer les suites de mêmes couleurs les remplaçant par le nombre de répétitions. (Par exemple, 12 12 12 12 04 12).

Le GIF :

Caractéristiques : GIF signifie "Graphic Interchange Format". Ce format a été créé par Compuserve Inc., qui à sorti 2 versions : le GIF87a (Mai 87) et le GIF89a. Il ne supporte que 256 couleurs, c'est pourquoi il utilise le modèle RIG et les palettes. La taille maximale de l'image est 2^{16} x 2^{16} pixels. Bien qu'il ne contienne qu'une image, il peut en supporter plusieurs par fichier.
Le GIF 89a reprend les mêmes caractéristiques que la précédente version, en ajoutant une possibilité de transparence de couleur, d'animation d'images, et d'effet d'entrelacement. Ces dernières visent à le rendre format standard du web.

Le format GIF est particulièrement approprié aux images dotées d'un nombre réduit de couleurs ou aux surfaces d'une seule couleur: boutons, traits, illustrations. Il est indispensable lorsqu'une partie de l'image doit être transparente ou pour créer une petite animation. Par contre, en raison de sa palette, il n'est pas adapté aux photos ou aux graphiques avec des dégradés de couleurs. (Voir JPEG)

Compression : Il utilise la compression LZW (du nom de ses inventeurs : Lempel, Ziv, Welch). Tous les fichiers GIF sont compressés, et tous avec ce même algorithme.

Remarque : La compression LZW appartient à Unisys. Il est breveté, et fait l'objet de limitations dans son usage. Dernièrement, la société a réclamé des droits sur l'utilisation de son fichier. Il faut donc payer une licence pour diffuser des images GIF sur Internet (à moins de les avoir édité avec un produit ayant acquis la licence).

Le JPEG :

Caractéristiques : JPEG signifie "Joint Photographic Expert Group". (Et son extension est ".jpg", et non ".jpeg"). Comme le BMP, il peut contenir jusqu'à 16 millions de couleurs. C'est pourquoi il est préféré pour les photos. Ses performances sont en moyenne les meilleures de tous les formats, car il supprime les éléments de l'image qui ne sont pas perceptibles par un œil humain. En revanche, il ne supporte pas la transparence.

Compression : Pour résumer les étapes de la compression, on pourrait dire {(Conversion Luminance/Chrominance) – (DCT (Transformée en cosinus discret)) – (Quantification) – Algorithme de Huffman)}. Le rapport taille/qualité est choisi par l'utilisateur. L'optimisation des images JPEG est toujours un compromis entre qualité et taille de fichier.

Le PNG :

Caractéristiques : Le format PNG (se prononce "Ping"), qui signifie "Portable Network Graphics", est une "variante" du GIF. En effet, il a été créé pour surpasser les limitations du GIF, et en même temps de distribuer un format libre de droits.
Il supporte en plus une transparence améliorée. La couche alpha (qui donne un effet plus ou moins transparent) peut être codée sur 8 ou 16 bits, contre seulement 8 pour le GIF. La principale amélioration est le support de couleurs en 24 bits. De plus, les résultats montre qu'il est de 10 à 30% plus efficace que le GIF.
Des tests montrent qu'il supporte très bien les dégradés, ce qui fait de lui le format idéal pour les images simples et moyennement complexes. C'est donc le format du futur, qui a tendance à être de plus en plus présent sur les sites Web.
Toutefois, il n'est supporté que par les dernières versions des navigateurs Internet.

Compression : Il utilise la compression LZ77, proche de la LZW, mais non brevetée. Il est ainsi un format gratuit.

Bilan :

Ces 4 formats étudiés révèlent la présence d'algorithmes très différents. Huffman, LZW et RLE sont les principaux. Il faut voir ensuite comment ils fonctionnent.

Séance du 04/10/01

08h00 – 09h50 : Salle informatique D09

Il m'avait semblé que la compression par dictionnaire était la LZW. En fait, cette compression se base sur les premières version, sorties plus tôt.

Il y avait eu auparavant la LZ77, LZ78, puis en 1984 LZW.

L'algorithme LZ77

En 1977, Abraham Lempel et Jacob Ziv publient l'algorithme à dictionnaire LZ77 qui relance la recherche en matière de compression.

L'algorithme utilise le principe d'une fenêtre coulissante de longueur ***N*** caractères, divisée en 2 parties, qui se déplace sur le texte de gauche à droite. La seconde partie, qui est la première à rencontrer le premier caractère du texte, contient ***F*** caractères : c'est le tampon de lecture. La première partie, alors égale à (***N*** **-** ***F***) sera appelée ***D***. C'est le dictionnaire.

Initialement, la fenêtre est située ***D*** caractères avant le début du texte, de façon à ce que le tampon de lecture soit entièrement positionné sur le texte, et que le dictionnaire n'y soit pas. Les ***D*** caractères sont alors des espacements.

A tout moment, l'algorithme va rechercher, dans les ***D*** premiers caractères de la fenêtre, le plus long facteur qui se répète au début du tampon de lecture. Il doit être de taille maximale ***F***. Cette répétition sera alors codée par le triplet (***i, j, c***) :

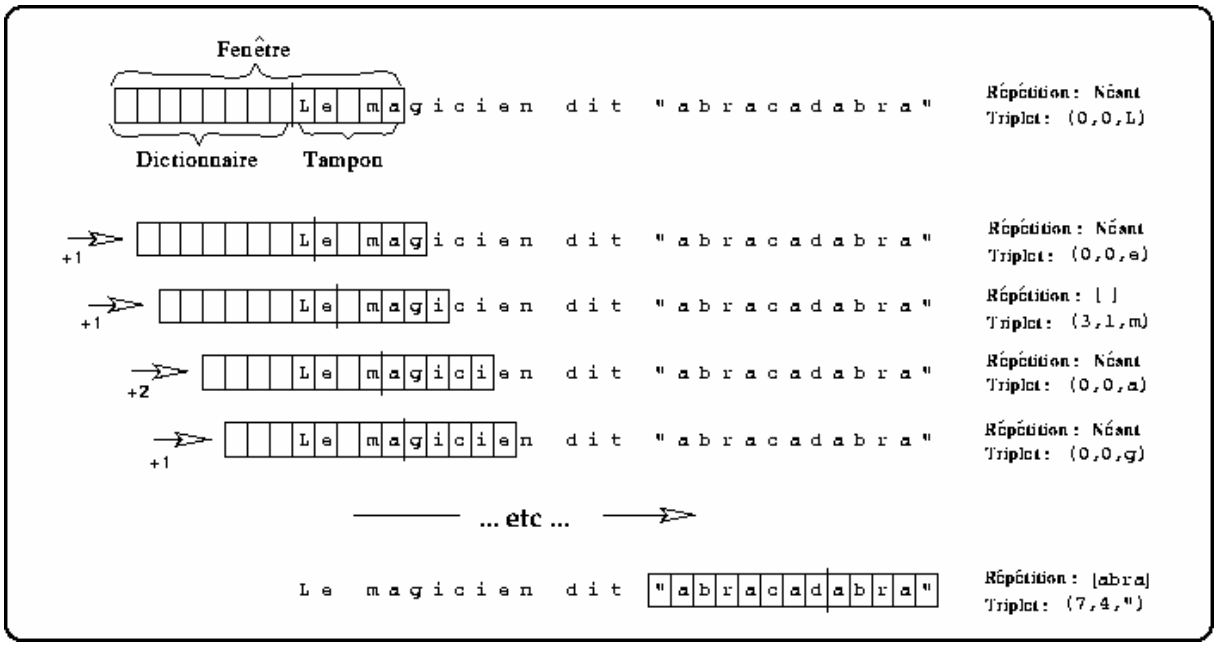
- ***i*** est la distance entre le début du tampon et la position de la répétition dans le dictionnaire.
- ***j*** est la longueur de la répétition.
- ***c*** est le premier caractère du tampon différent du caractère correspondant dans le dictionnaire.

La répétition peut chevaucher le dictionnaire et le tampon de lecture.

Après avoir codé cette répétition, la fenêtre coulisse de ***j*****+1** caractères vers la droite. Le codage du caractère ***c*** ayant provoqué la différence est indispensable dans le cas où aucune répétition n'est trouvée dans le dictionnaire. On codera alors (0, 0, ***c***).

Exemple : Phrase à compresser : Le magicien dit "abracadabra".
Taille de la fenêtre : ***N*** = 13, ***D*** = 8, ***G*** = 5.
(Voir *schéma sur la feuille suivante*)

Explications : Au point de départ, aucune répétition n'est trouvée puisque le dictionnaire ne contient que des espacements. Le codage des deux premiers caractères du texte est alors (0,0,**L**) (0,0,**e**), la fenêtre se déplaçant à chaque fois d'un caractère vers la droite. Ensuite l'espacement est rencontré. Comme il est déjà présent dans le dictionnaire 3 caractères avant le début du tampon, et puisque le caractère **m** termine la répétition, le triplet est (3,1,**m**). On continue alors: (0,0,**a**) , (0,0,**g**) …



On continue ainsi jusqu'à la fin de la séquence à coder.

Le tampon de lecture arrive finalement alors **abra"**. Le plus long mot présent dans le dictionnaire et commençant le tampon contient 4 caractères, c'est **abra**. Dans le dictionnaire, il se situe 7 caractères avant le début du tampon. Le caractère du tampon marquant la différence avec le facteur correspondant du dictionnaire est "

Le triplet est donc (7,4,").

A la fin, on obtient la suite de tous les triplets à coder : (0,0,**L**) (0,0,**e**) (3,1,**m**) (0,0,**a**) (0,0,**g**) (0,0,**i**) (0,0,**c**) (2,1,**e**) (0,0,**n**) (0,0,) (0,0,**d**) (5,1,**t**) (4,1,") (0,0,**a**) (0,0,**b**) (0,0,**r**) (3,1,**c**) (2,1,**d**) (7,4,").

On code ces triplets en binaire, en associant une valeur fixe pour chaque triplet

On remarque que le codage est d'autant plus intéressant que le mot répété est long. Pour avoir des chances d'obtenir de longues répétitions, le dictionnaire doit être de taille suffisante (de l'ordre de plusieurs milliers de caractères).

La décompression est très simple et rapide. A partir de la suite de triplets, le décodage s'effectue en faisant coulisser la fenêtre comme pour le codage. Le dictionnaire est donc reconstruit de gauche à droite en une seule fois.

L'algorithme comprime les informations au fur et à mesure du déplacement de la fenêtre. Contrairement à Huffman, il n'a besoin que d'un seul passage sur la chaîne à coder. Il peut donc être utilisé sur des données en transit. C'est pourquoi la norme V42bis des modems utilise la compression LZ77 pour réduire la quantité de données à télécharger. Il faut bien sûr que la mémoire du modem soit suffisante pour contenir la fenêtre de l'algorithme.

On peut ainsi dépasser la valeur maximale indiquée par le constructeur (33,6 Kbps, 56 Kbps…).

L'algorithme LZ78

Il est tout simplement une amélioration du LZ77 par les mêmes auteurs.

La fenêtre coulissante a disparu. Le dictionnaire est toujours présent, mais séparé du pointeur. Le pointeur lit le fichier de gauche à droite.

Au point de départ, le dictionnaire ne contient aucun facteur. Les facteurs sont numérotés à partir de 1. A tout moment, l'algorithme recherche le plus long facteur du dictionnaire qui concorde avec la suite des caractères du texte. Il suffit alors d'encoder le numéro de ce facteur et le caractère suivant du texte : (***num***, ***c***). Le caractère ***c*** concaténé au facteur numéro ***num*** nous donne un nouveau facteur qui est ajouté au dictionnaire pour être utilisé comme référence dans la suite du texte.

Exemple : Codage de la chaîne **aaabbabaabaaabab**

Parcours du texte:	**a**	**aa**	**b**	**ba**	**baa**	**baaa**	**bab**
Numéro de facteur:	1	2	3	4	5	6	7
Codage:	(0,a)	(1,a)	(0,b)	(3,a)	(4,a)	(5,a)	(4,b)

Au point de départ, le dictionnaire ne contenant aucun facteur, aucune concordance ne peut être trouvée entre les caractères du début du texte et un facteur. On code donc (**0**,**a**) ce qui signifie le facteur vide suivi du caractère **a**. Ce nouveau facteur portera le numéro **1** dans le dictionnaire.

Ensuite, **aa** correspond au premir facteur du dictionnaire suivi de **a**. on code donc (**1**,**a**). Ce nouveau facteur est numéroté **2**.

Les caractères suivants (**bba** ...) sont inconnus dans le dictionnaire. On codera donc que l'on a le facteur vide suivi du caractère **b** : (**0**, **b**). Ce nouveau facteur porte le numéro **3**. Suivent alors les caractères **ba** considérés comme le facteur **3** suivie du caractère **a**, et ainsi de suite...

Comme pour LZ77, LZ78 comprime les informations au fur et à mesure de leur réception.

Le décodage est très simple également parce qu'il suffit au décodeur de reconstruire le dictionnaire au fur et à mesure du décodage. Les numéros des facteurs seront donc les mêmes que pour le codage et les facteurs pourront être interprétés sans problème.

L'algorithme LZW

L'avantage de LZ78 par rapport à LZ77 était de passer de 3 informations à coder à 2. Terry Welch, en 1984, publie son travail, complément de l'algorithme LZ. Il est parvenu à n'avoir à coder plus qu'une seule information.

Le dictionnaire est déjà rempli de 255 caractères. En effet, les caractères du code ASCII sont déjà présents au début du dictionnaire. En parcourant la chaîne à coder, on enregistre donc le premier élément en tant que n° 256.

Ici, la chaîne à coder est : **"AIDE TOI LE CIEL T AIDERA"**

étape	lu	émis (déc)	émis (bin)	tampon	adresse	séquence
1	A			A	0...255	ascii 0...255
2	I	65	0100 0001	AI	256	AI
3	D	73	0100 1001	ID	257	ID
4	E	68	0100 0100	DE	258	DE
5	blanc	69	0100 0101	E blanc	259	E blanc
6	T	32	0010 0000	blanc T	260	blanc T
7	O	84	0101 0100	TO	261	TO
8	I	79	0100 1111	OI	262	OI
9	blanc	73	0100 1001	I blanc	263	I blanc
10	L	32	0100 0000	blanc L	264	blanc L
11	E	76	0100 1100	LE	265	LE
12	blanc	*SP*	1111 1111	E blanc		
13	C	259	1 0000 0011	E blanc C	266	E blanc C
14	I	67	0 0100 0011	CI	267	CI
15	E	73	0 0100 1001	IE	268	IE
16	L	69	0 0100 0101	EL	269	EL
17	blanc	76	0 0100 1100	L blanc	270	L blanc
18	T			blanc T		
19	blanc	260	1 0000 0100	blanc T blanc	271	blanc T blanc
20	A	32	0 0010 0000	blanc A	272	blanc A
21	I			AI		
22	D	256	1 0000 0000	AID	273	AID
23	E			DE		
24	R	258	1 0000 0010	DER	274	DER
25	A	82	0 0101 0010	RA	275	RA
26		65	0 0100 0001			

La colonne 1 numérote les étapes. La seconde montre l'action du pointeur (lecture). On se réfère ensuite à la colonne 5 pour y étudier le tampon, et le comparer avec le dictionnaire (colonnes 6 et 7). Le résultat est ensuite émis (colonnes 3et 4).

L'objectif de l'algorithme LZW est de construire un dictionnaire où les séquences seront désignées par une adresse dans le dictionnaire.

Les premières étapes du compactage sont les suivantes : le premier octet lu est ***A***. l'algorithme n'effectue que cette action à la première étape. ***A*** est mémorisé dans un *tampon T*. Puis l'octet ***I*** est lu. On le place dans le tampon T qui contient alors ***AI***. On regarde si la chaîne de *T* existe dans le dictionnaire. Elle n'existe pas, donc on ajoute la chaîne de *T* (***AI***) dans le dictionnaire et on émet le code de ***A*** à défaut d'une chaîne plus longue. Ensuite on retire ***A*** de T et on recommence le processus.

Si la chaîne dans *T* existe dans le dictionnaire, on n'émet rien dans le fichier de sortie, mais on lit le caractère suivant pour rechercher une chaîne plus longue. C'est ce qui se passe à l'étape 12 : ***E blanc*** est déjà dans le dictionnaire, on tente de chercher ***E blanc C*** à l'étape suivante. La recherche de ***E blanc C*** échoue et on émet le code de ***E blanc*** puis on garde seulement ***C*** dans *T* (le dernier caractère lu) : On se retrouve comme en phase d'initialisation.

On peut remarquer qu'à l'étape 12, au lieu de ne rien faire on a émis un code ***SP*** = 11111111. Ce code signifie qu'à partir de ce point les adresses ne seront plus émises sur 8 bits mais sur 9 bits, ceci pour permettre de gérer des adresses de tailles variables. Ce code ne sert qu'au décompacteur pour savoir l'action qu'il doit faire, en plus du décodage. D'autres codes peuvent être émis, comme Vider le dictionnaire. Le code ***SP*** peut être codé sur 17 bits (8 puis 9) pour ne pas le confondre à la décompression avec le caractère 255.
C'est cette compression LZW qui est utilisée dans les images GIF.

Séance du 11/10/01

08h00 – 09h50 : Salle informatique D09

Les compressions LZ** de la dernière séance ayant été très intéressants, j'ai décidé de continuer sur ce sujet. Et il existe encore des variantes des compressions LZ :

- LZP : Algorithme qui se base sur la répétition de phrases entières.
- LHA, LZS, LZX, LZH : Algorithmes quasiment identiques, encore dérivés du LZ77. Il n'est employé que pour l'utilitaire Lharc, très répandu au Japon, mais de moins en moins utilisé dans le Monde.
- LZSS : Développé par James Storer et Thomas Szymanski en 1982, à partir des travaux de Lempel et Ziv. Les éléments répétés se présentent sous forme d'une arborescence, ce qui rend le travail du compresseur plus rapide lors de recherches dans le dictionnaire. Il utilise aussi moins de caractères pour stocker l'information compressée, d'où le gain d'octets.

J'ai aussi appris que les algorithmes LZ77 et LZSS sont dits "*algorithmes à fenêtre coulissante*" alors que LZ78 et LZW sont dits "*algorithmes à dictionnaire*".

Il existe aussi 2 autres codages, mais peu connus et peu utilisés :

- **Codage topographique**

Ce type de compression est utilisable surtout dans les fichiers textes. On sélectionne un octet fréquemment utilisé dont on va privilégier le compactage. Un octet topographique indique ou non la présence d'information dans les octets suivants.

Ex :
ABAGABDA = 8 octets
(01010110)BGBD = 5 octets
L'octet favorisé est le A

L'utilisation de cette méthode est assez limitée, on la trouve parfois pour les fichiers textes, où l'on favorise l'espace.

- **Codage relatif**

Le codage relatif est utilisé pour la compression de données numériques, pour les résultats d'expériences. On résume les bits qui ne varient pas en début de séquence, puis on précise que les bits qui évoluent.

Ex :
00101010 00101110 00101001 0010111
#5 00101 4 010 110 001 111

On remarque que même avec des exemples il est difficile de gagner de l'espace.

Séance du 18/10/01

08h00 – 09h50 : Salle informatique D09

Durant cette séance, j'ai cru trouver de nouveaux algorithmes, ou codages, et des compléments des méthodes LZ**. Malheureusement, après avoir réexaminé le contenu des pages HTML enregistrées, je me suis aperçu que de nombreux sites illustraient la compression à fenêtre coulissante avec le même schéma. Les explications étaient les mêmes, sensiblement modifiées. On y trouve même le site d'un élève de 1°S, lui aussi étudiant les compressions graphiques en TPE, qui explique sommairement les différents algorithmes. Mais le contenu et les images sont tirés de quelques sites voisins, alors que d'après lui, il avait "*passé environ 3 heures à convertir proprement* [son] *TPE du format MS Word en HTML 4.0 avec Notepad*".

Le bilan est finalement bien maigre par rapport aux autres séances. J'ai toutefois pu améliorer mon vocabulaire :

- ✓ Une compression symétrique est une compression où le même algorithme est utilisé pour la compression et la décompression (compression asymétrique , souvent plus longue à la compression).
- ✓ Il existe les codages non-adaptatifs (qui contiennent un dictionnaire statique de caractères prédéfinis), les codages adaptatifs (qui construisent leurs propres dictionnaires au fur et à mesure de la compression) et les codages semi-adaptatifs (qui sont un mélange des deux méthodes). La compression LZW est adaptive, car le dictionnaire construit pour compresser n'est pas sauvegardé (gain énorme de place). Il est restitué de la même façon à la décompression. Cyril, dans le JPEG, avait déjà vu cela. L'arbre de Huffman est adaptatif.

Nous avons revu avec Cyril les thèmes de notre TPE. Alors qu'il va poursuivre sur le JPEG, je pense que les compressions LZ77, LZ78 et LZW demandent un enregistrement en bits. Et je ne manie pas du tout l'utilisation des bits. Je vais donc me tourner vers la compression RLE.

Séance du 25/10/01

08h00 – 09h50 : Salle informatique D09

Alors que nous étions tous les deux ce matin sur les ordinateurs, nous nous aperçûmes qu'une méthode qui nous était déjà passée sous les yeux, sans conséquences, se révèle être intéressante. Cette méthode, le codage prédictif, fonctionne sur les images. Comme son nom l'indique, il effectue une prédiction. Étant donné que des blocs d'une image ont tendance à varier d'une manière progressive, c'est à dire que deux blocs voisins ont de sérieuses chances d'être similaires, on peut essayer, à partir de quelques pixels, d'imaginer ce que pourraient être les pixels voisins. Par exemple, sur une image 24 bits, un pixel (j, i) dont la valeur Rouge est 200, et un pixel (j, i+2) dont la valeur Rouge est aussi 200, on peut supposer que le pixel (j, i+1) possède lui aussi une valeur Rouge égale à 200. On calcule ensuite la différence entre la couleur du pixel à venir et la valeur de sa prédiction. Cette différence est donc une valeur non plus absolue, mais relative. A la place d'utiliser 24 bits pour coder une couleur, on va seulement coder une différence, valeur qui occupe rarement plus de 24 bits.

Avec un schéma, on obtient cela :

.	.	.	.	.	.	.	.
.	A	B	C	.	.	.	.
.	D	X					

Le pixel à coder est X. A l'aide des pixels l'entourant (dans le sens de lecture, donc pixels en haut et à gauche), on effectue des petits calculs pour "prédire" la valeur de X. Le site nous donne des indications :
(A + B + C + D) / 2
(2B + 2D + A + C) / 6
(B + D) / 2 …

Il suffit de calculer la différence entre la valeur du Rouge prédit et la valeur du Rouge réelle, entre la valeur du Vert prédit et la valeur du Vert réelle, et enfin entre la valeur du Bleu prédit et la valeur du Bleu réelle.

Cyril m'a dit qu'il allait s'intéresser de plus près à l'implémentation de ce codage avec Delphi.

J'ai également compris que le codage des couleurs n'était pas le même dans tous les langages. En effet, j'avais vu le rouge codé FF 00 00, puis ensuite 00 00 FF.

En fait, c'est tout simplement que Windows utilise le codage RGB en codant les couleurs suivant la suite Rouge - Vert - Bleu. L'autre codage, c'est en HTML. C'est donc pour cela qu'il faut faire attention aux valeurs des couleurs rencontrées.

Séance du 08/11/01

08h00 – 09h50 : Salle informatique D09

Pendant les 2 semaines durant lesquelles nous n'avons pas eu TPE, nous avons regardé les conséquences de la méthode prédictive, découverte le 25/10.

Il fallait faire un petit calcul pour trouver le pixel par prédiction. Après plusieurs essais en comparant la taille des fichiers générés, nous avons privilégié la formule (B + D) / 2 pour un pixel au centre de la matrice. Si X est sur la première colonne, on prend seulement B en compte. Si X est sur la première ligne, on prend seulement D en compte. Cette solution donne le meilleur taux de compression, et est une des plus simples.

Après la prédiction, on applique une compression RLE. Si plusieurs pixels sont identiques, les différences Prédiction/Valeur seront identiques. La compression RLE est spécialisée dans les éléments redondants, donc on l'utilise ici. Ensuite enregistre les valeurs grâce à l'arbre de Huffman, que Cyril avait bien étudié pour le JPEG. La compression est donc conservative.

A part ces résultats, il fallait commencer avec la compression RLE. Cette dernière paraît simple : coder les octets redondants. J'ai donc construit avec Delphi une application qui ouvrait les fichiers et qui compressait les caractères. Le fichier sélectionné était ouvert dans un composant RichEdit. Le programme lisait le nombre de caractères identiques consécutifs, puis enregistrait le nombre d'itérations, suivi du caractère en question. Les résultats était médiocres : les taux avoisinaient les 200%. Par exemple, on enregistrait "bonjour" en "1b 1o 1n 1j 1o 1u 1r". Et dans les fichiers textes, les caractères redondants sont rares. Pour les fichiers non textes, je disposais toujours du composant RichEdit.
Plusieurs problèmes se posèrent :
- Le composant RichEdit n'affiche pas le contenu réel du document, à la manière d'un éditeur décimal/hexadécimal. De même, pour le problème était identique pour l'enregistrement du fichier traité.
- Les variables ne pouvaient pas supporter le contenu des fichiers. L'application plantait souvent.

Ce premier contact avec la compression RLE nous fit vérifier que cette compression, certes conservative, est inefficace pour traiter les fichiers non graphiques.

Donc durant les prochaines séances, il faudra que je remette la méthode sur une image.

Séance du 15/11/01

08h00 – 09h50 : Salle informatique D09

Après les désillusions de la semaine dernière, il fallait que je puisse ouvrir une image avec Delphi pour la compresser ensuite. J'ai décidé de ne pas utiliser le composant TImage fourni, car il peut aussi effectuer ces tâches. J'ai donc décortiqué la structure d'un fichier Bitmap.

D'un point de vue général, il est constitué de 4 parties :

- Entête :
  - Entête du fichier : "FileHeader"
  - Entête du bitmap : "BitMapHeader"
- Palette (optionnelle)
- Corps de l'image

En détail, l'entête du fichier donne :

- Signature : "BM", ou "42 4D" (2 octets)
- Taille du fichier en octets (4 octets)
- Espace réservé : octets égaux à 0 (4 octets)
- Offset de l'image : adresse par rapport au début du fichier du premier octet du bitmap (4 octets)

Et l'entête du bitmap :

- Taille de l'entête (4 octets)
- Largeur de l'image (en pixels) (4 octets)
- Hauteur de l'image (en pixels) (4 octets)
- Nombre de plans utilisés : toujours égal à 1 (2 octets)
- Nombre de bits par pixels (2 octets)
- Méthode de compression : 0 pour une image non compressée, 1 ou plus pour une image compressée (4 octets)
- Taille de l'image compressée : Elle est donc égale à (Hauteur x Largeur x 3), ou encore à (Taille du fichier – 54) lorsque l'image n'est pas compressée. (4 octets)
- Résolution horizontale (pix./mètre) (4 octets)
- Résolution verticale (pix./mètre) (4 octets)
- Nombre de couleurs de la palette (4 octets)
- Masque (4 octets)

Ensuite, on trouve l'image. Elle est simplement enregistrée avec 3 couleurs par pixels (24 bits). Sur les images 4 ou 8 bits, on a besoin d'une palette, qu'il faut enregistrer au début du fichier. Mais on ne sait pas comment lire une palette, alors nous sommes obligés de n'utiliser les images 24 bits.

On trouve donc après l'entête, au $55^{ème}$ octet, la composante bleue du premier pixel, puis la verte, et enfin la rouge. (En fait, le code HTML respecte la suite RGB, tout comme Delphi. Mais c'est Windows et ses Bitmaps qui inversent l'ordre).

Après plusieurs essais (ratés), on s'est aperçus que l'image est enregistrée en commençant par le pixel en bas à gauche. Il faut donc retourner l'image, ou plus simplement décoder en commençant par le bas.

J'ai regardé les informations sur le fichier GIF que l'on avait trouvées. L'entête est un peu expliqué, mais des points importants restent obscurs. De plus, l'image bitmap nous ayant déjà donné du fil à retordre, nous n'allons pas continuer sur l'étude de la structure des autres fichiers.

Séance du 22/11/01

08h00 – 09h50 : Salle informatique D09

J'ai réussi à lire une image bitmap avec Delphi sans le composant TImage. Il me fallait maintenant des informations spécifiques à la compression RLE. Le site compressions.multimania.com donnait une méthode, en indiquant que c'était la méthode conventionnelle utilisée par Windows. On posera 1 pixel de l'image = 1 octet du fichier.

Si un pixel est répété 3 fois ou plus, on écrit le nombre d'itérations suivi de sa valeur		
Exemple	07 07 07 07 07	05 07
Exemple	0B 0B 0B 0B 0B 0B 0B 0B 0B 0B	0A 0B
Sinon, la suite n'est pas codée. On écrit 00, puis le nombre de valeurs, puis ces valeurs		
Exemple	FB FB 89 23	00 04 FB FB 89 23
Si cette suite est impaire, on rajoute 00 à la fin		
Exemple	23 65 55 34 22	00 05 23 65 55 34 22 00

Il existe des codes spéciaux :

00 01	Fin de ligne
00 00	Fin de bitmap
00 02 XX YY	Déplacer le pointeur dans l'image de XX colonnes et YY lignes (très peu utilisé)

Donc pour enregistrer un fichier bitmap compressé avec la compression RLE, il faut écrire l'entête classique de 54 octets (en ajustant les octets 30 à 34 en fonction de la compression), puis sauvegarder l'image bitmap en suivant la procédure ci-dessus.

D'autres sites proposent des méthodes différentes, en codant à partir de 4 répétitions, d'autres encore incluent un caractère spécial.

Séance du 13/12/01

08h00 – 09h00 : SalleD19

Nous avons exposé brièvement l'avancement de nos travaux ainsi que les autres groupes de la classe.

09h00 – 09h50 : Salle informatique D09

Cette semaine, j'ai appliqué la RLE aux fichiers Bitmaps 24 bits. En un seul passage sur l'image, les octets redondants sont repérés et codés. Le programme procède ligne par ligne.

J'ai fait des modifications concernant les informations trouvées sur Internet. En sortie de l'algorithme, on ne trouve que 2 informations différentes : des pixels redondants, ou des pixels non redondants. Tous doivent pouvoir être différenciables par le décompacteur. D'où l'utilisation de signaux. En effectuant plusieurs essais, et en comparant le gain de compression et le temps dépensé, j'ai finalement attribué les codes suivants :

- Si ils se répètent, on écrit le nombre de répétitions, puis le pixel à répéter.
- Si des pixels consécutifs ne se répètent pas, on écrit 00, suivi du nombre de pixels, suivi de ces pixels. On ne fait pas la différence entre les nombres pairs et impairs de pixels.
- Les codes 00 00 et 00 01 pour la fin de ligne et fin de fichier (codage conventionnel) sont supprimés. Puisque l'on connaît la hauteur et la largeur du fichier, l'algorithme de décompression change de ligne automatiquement.
- J'ai inséré les codes 01 et 02, qui étaient inutilisés (car le 00 signale les octets non redondants, et on ne code les autres qu'à partir de 3 répétitions (donc de 03 à 255)). Le 01 indique qu'il faut ajouter 256 à l'octet suivant pour répéter le pixel. On l'utilise si les pixels redondants sont compris entre 255 et 511. On ne rencontre déjà pas souvent plus de 500 pixels identiques consécutifs. Et pour le cas très rare ou plus de 511 pixels seraient alignés, on écrit 02, qui signale que le nombre de redondances est codé sur les 2 octets suivants. On trouve ensuite sur 3 octets la couleur à reproduire.

Contrairement au principe Microsoft qui lit et écrit les fichiers de bas en haut, le programme effectue la compression plus simplement. Si l'on appelle (0, 0) le pixel en haut a gauche, (0, 1) le pixel immédiatement à sa droite, "L" la largeur de l'image décrémenté de 1, et "H" la hauteur de l'image décrémentée de 1, le sens de lecture est alors de (0, 0) à (0, L), puis de (1, 0) à (1, L) … jusqu'à (H, L).

Naturellement, à la décompression, on replace les pixels dans le même ordre.

Séance du 20/12/01

08h00 – 09h50 : Salle informatique D09

J'ai continué le programme de compression RLE, qui fonctionnait plutôt bien jeudi dernier. Mais entre-temps, j'ai essayé avec des bitmaps plus colorés et plus grands. Donc le fichier de sortie était plus gros. Le problème, c'était que le tableau temporaire dans lequel sont déposés les octets (destinés à être enregistrés dans un fichier avec les fonctions FileWrite) doit être statique (car les fonctions Borland ne supportent pas de tableaux dynamiques). Et donc je devais déclarer un tableau statique capable de contenir des très grands bitmaps. Ce qui est impossible au-delà d'une certaine limite, sinon l'erreur *EStackOverflow* (qui est déclenchée quand la pile du thread en cours déborde dans la page de garde finale, c'est-à-dire quand le système ne peut plus agrandir dynamiquement la pile. Cela peut être provoqué par des variables locales très volumineuses, des appels récursifs de routines très profonds ou du code assembleur incorrect.) est déclenchée.

La solution est d'enregistrer les octets compressés petit à petit. J'ai décidé de le faire à chaque fin de ligne. La taille du tableau est de 3001 éléments. Ce qui me limite à des images de 2999 pixels de long sans une seule répétition. Bien sûr, si le taux de compression de la ligne est de 50%, je peux enregistrer dans ce tableau 6000 pixels environ. Des images Bitmap, de 6000 pixels de large étant rarement trouvables, je suppose que le programme fonctionnera dans tous les cas. (dans mes essais, une image de 1000x1000 pixels est compressée sans problèmes).

En voulant voir les capacités de la compression RLE, j'ai remplacé la constante de redondance (nombre à partir duquel des octets redondants sont codés comme tels) par une variable. Et j'ai constaté que si elle était à 2, le gain de compression est plus élevée qu'avec 3 (ancienne valeur). Mais pour utiliser cette valeur, il me faut modifier le système de codage. En effet, le code 02 (pour signaler que les octets redondants sont supérieurs à 511) n'est plus valable, car il est utilisable. J'ai donc remplacé la fonction du code 01 par celle du code 02, et supprimé ce dernier. Ainsi, nous avons :

- "00" suivi de "NbPixels" pour indiquer que les NbPixels * 3 octets suivants correspondent à des pixels non redondants.
- "NbRep" suivi de "Pix" pour répéter NbRep fois le pixel Pix.
- "01" suivi de "Nb1", "Nb2", "Pix" pour répéter (Nb1 * Nb2) fois le pixel Pix.

J'ai aussi cherché à améliorer l'algorithme. Je l'ai adapté à une compression verticale : le programme compresse de différentes façons et affiche le résumé à la fin. On a ensuite la possibilité de sauvegarder l'image dont le gain de compression est le plus élevé. J'ai d'autres idées de compression dans d'autres sens, mais que j'étudierai plus tard.

Tout comme je format JTPE avec le Jpeg, le format RTPE est notre format qui stocke une image compressés avec l'algorithme RLE. Son entête est (provisoirement) de la forme :

"RTPE" (4 octets)
Taille du fichier (3 octets)
Largeur de l'image (2 octets)
Hauteur de l'image (2 octets)
Type de compression (1 octet).

Séance du 17/01/02

08h00 – 09h50 : Salle informatique D09

En sortant de la salle de présentation jeudi dernier, j'ai eu envie d'exploiter les compressions à dictionnaire (LZ77, LZ78 et LZW).

D'après les résultats du format GIF et les performances des logiciels tels que WinZip ou WinRAR (qui utilisent principalement le GIF, algorithme conservatif actuellement le plus performant), je devrais obtenir des bons taux de compression.

Je laisse donc pour le moment de coté les variantes de la RLE que j'avais prévues d'implémenter.

J'ai bien envie de traiter les trois compressions, LZ77, LZ78 et LZW, et d'ainsi distinguer les améliorations dans l'algorithme et dans les performances. Mais pour être sûr que au moins une des 3 soit complétée avant la date de passage, je vais commencer par celle, actuellement, qui est la plus performante et la plus utilisée : LZW.

J'ai commencé de la même façon que pour la RLE, de façon a bien cerner les mécanismes de l'algorithme. J'ai placé 2 composants TEdit, et essayé de compresser la chaîne de caractères entrée dans le champ.

Le principe est plus compliqué que pour la RLE. Il faut construire un dictionnaire. Une autre difficulté est le traitement par bits. Le langage Pascal n'étant pas adapté pour traiter des données de taille inférieure à l'octet, j'ai emprunté les fonctions "LireBit" et "EcrireBit" que Cyril avait composé, qui permettent d'accéder à un fichier bit par bit.

La tâche fut plus hardie que pour la compression précédente, mais je peux tirer mes premières conclusions, qui concordent avec les informations publiées sur Internet :

Pour que la compression soit réellement efficace, il faut d'abord construire un dictionnaire assez volumineux. Par exemple, la compression de "AVOGADRO" (64 bits) donne "AVOGADRO" (64 bits) : Le dictionnaire à enregistré les valeurs {AV, VO, OG, GA, AD, DR, RO} mais ne les a pas réutilisées.
Le compression de "EINSTEIN" (64 bits) donne "EINST" + SP + Code256 + "N", ce qui fait (5 * 8) + (8 + 9) + 9 + 9 = 75 bits. La lettre "N" finale est codée sur 9 bits, car une valeur du dictionnaire a déjà été utilisée auparavant. Toutefois, les lettres 6 et 7 ("EI") occupent 9 bits. On voit donc ici l'importance d'avoir un dictionnaire bien rempli. (On remarque aussi que l'algorithme n'a pas repéré la suite "EIN").

Séance du 24/01/02

08h00 – 09h50 : Salle informatique D09

J'ai essayé de passer l'algorithme sur une image. Mais je me suis heurté au problème du dictionnaire : En résumé, sur du texte, la compression LZW consiste à lire un ensemble de caractères, et s'il n'appartient pas au dictionnaire, on l'inscrit. Sinon, on émet son code.
Si l'on essaye avec des pixels : il faut entrer dans le dictionnaire des suites de pixels. (J'ai essayé avec des images Bitmaps 24 bits). Or un pixel est déjà un ensemble de 3 octets. Donc il faut entrer des suites de suites d'octets, ce qui fait construire un dictionnaire à 3 dimensions. Mais ensuite, lorsqu'on passe sur un second ensemble, il faut le comparer aux ensembles déjà entrés dans le dictionnaire. La comparaison est donc compliquée et très longue. J'ai essayé de remplacer les 3 octets par un TColor. Mais lorsqu'il s'agit d'assembler les TColors pour comparer la suite à une autre, je n'ai pas réussi à programmer cela avec Delphi. J'ai perdu beaucoup de temps à essayer, et compris que même si le programme fonctionnait ensuite, il serait très lent. D'ailleurs, le format GIF ne fonctionne qu'avec des images de 256 couleurs maxi, soit 1 pixel = 1 octet, ce qui simplifie beaucoup la tâche. Mais l'équivalence 1 pixel = 1 octet est la même dans le fichier directement.

J'ai donc décidé de recommencer, mais en traitant directement le fichier.

Je traite le fichier à l'aide des procédures FileOpen, FileRead, FileSeek …

Je me suis posé une nouvelle fois au problème du dictionnaire : la première fois, j'avais créé un tableau de *string*, car un *string* est défini comme une suite de *byte*. Ce à quoi je n'avais pas pensé, c'est que un *string* se termine par le caractère 0. Or, les caractères 0 sont très présents dans les images (00 = Absence de couleur dans le 8 bits (-> blanc) ou absence de composante rouge, verte ou bleu dans le 24 bits).

Finalement, j'ai déclaré un tableau de tableaux de *byte*.

Le programme est fonctionnel pour les fichiers de moins de 4096 octets. En effet, étant donné que les procédures de lecture et d'écriture dans un fichier nécessitent des tableaux statiques, le programme lit d'abord 4096 octets du fichier d'entrée, les traite, écrit le résultat dans le fichier de sortie. Ensuite, il lit les 4096 octets suivants du fichier d'entrée, les traite, et les écrit dans le fichier de sortie, au bout des caractères précédemment écrits.

Mais, avec la compression LZW, on calcule en bits. Or, on ne peut écrire que des octets. Donc il faut garder les bits en trop du premier passage, et les coller au début du second passage. Par exemple :
- Fichier d'entrée : octets 0 - 4096 => Compression => 16007 bits.
On écrit seulement 16000 bits (2000 octets) dans le fichier de sortie et on garde les 7 bits dans une variable. On recommence avec la suite du fichier d'entrée :
- Fichier d'entrée : octets 4097 - 8192 => Compression 12003 bits.
On insère les 7 bits restant du premier passage avant les 12003 bits. Ce qui fait 12010 bits à écrire. Mais on ne peut écrire que 1501 octets, et on place les 2 bits restants dans une variable.
- On continue … jusqu'à la fin où l'on écrit les bits restants sur un octet.

Le problème est que ce passage ne se réalise pas bien. Il faut trouver le bug !

Le programme ainsi formé traite non pas les images, mais les fichiers. Même si des fichiers peuvent être des images, il y en a de nombreux autres. On peut donc le comparer à un utilitaire de type WinZip.

Séance du 31/01/02

08h00 – 09h50 : Salle informatique D09

Depuis les derniers essais, je n'ai pas trouvé ce qui ne fonctionnait pas, mais j'ai trouvé d'autres petits bugs, et j'ai amélioré du code :

Il y avait un petit problème, que j'ai appelé "le bug du 000". En effet, compressons "000". Pour une meilleure compréhension, nous allons compresser "$0_1 0_2 0_3$", 3 caractères identiques : On entre dans le dictionnaire $0_1 0_2$ sous le code 260, et on émet 0_1. Ensuite en compare $0_2 0_3$ au dictionnaire. Il y est déjà, donc on émet le code 260. Ce qui donne en sortie {0, 260}. On remarque que le décompresseur ne peut pas comprendre ce code !. Il faut donc ajouter un test qui interdit une telle entrée !

Notre dictionnaire fait 4096 octets. Lorsqu'il est presque plein, et que nous avons une nouvelle chaîne à traiter, il faut la comparer à plus de 4000 autres chaînes !!! C'est le principal inconvénient de la compression LZW. J'ai modifié le dictionnaire : à la place de chaque chaîne, on inscrit la longueur de la chaîne suivi de cette chaîne. Ainsi, au lieu de comparer à toute la chaîne, on compare d'abord la taille de la chaîne à traiter avec les nombres inscrit dans les premiers emplacement du dictionnaire. Si les tailles concordent, on compare les chaînes. Sinon, on passe. Grâce à cet amélioration, j'ai divisé le temps de compression par 5 !. Ceci est du au fait que pour comparer une chaîne (suite d'octets), il faut comparer chaque octet de la suite. Il y donc des milliers de comparaisons par seconde. Le but des améliorations est de réduire ce nombre au strict minimum, sans négliger le moindre problème : la LZW est une compression conservative, et un bit qui change sur 2457600 dans un fichier exécutable de 300 Ko peut rendre le fichier illisible.

Ensuite, j'ai réussi à diviser le temps par 2, simplement en arrangeant le code :

```
Setlength(…);
If … then Exit;
```

De cette façon :

```
If … then Exit;
SetLength(…);
```

Le problème de la vitesse est très important. Le premier exemplaire utilisait 3 secondes pour compresser 3 octets. Le dernier prends 330 ms. Toutefois, le temps dépends considérablement des données à compresser : un fichier déjà compressé (avec WinZip) est quasiment à sa taille minimale. Mon programme ne peut rien y faire, mais il passe énormément de temps à comparer les suites. D'après certains sites, le meilleur moyen de faire des programmes rapides est de programmer en assembleur. Mais je préfère accepter le fait que mon programme soit un petit peu lent …

Le gain de compression n'est pas négligeable. Comme Cyril, à chaque amélioration de nos programmes, on compare avec les outils chers et populaires (PSP, WinZip …). Le format WTPE se trouve toujours à moins de 10% de WinZip, en gain de compression. Quand WinZip fait 45% sur un fichier texte, le programme fait 36%. Ces pourcentages varient beaucoup (mais ensemble !).

Séance du 21/02/02

08h00 – 09h50 : Salle informatique D09

Nous avons cherché des informations sur Internet qui pouvaient enrichir notre dossier, notamment sur les applications des algorithmes sur lesquels nous avons travaillé.

La compression est très utilisée dans les images satellites, par exemple pour les cartes topographiques : une carte nationale numérique au 1:25000 est fournie à une résolution de 500dpi et couvre une zone de 17 sur 12 km.

Pour couvrir la Suisse, il faut 260 images de 14000 x 9600 pixels. Actuellement ces images sont livrées en 8 couleurs, mais pour inclure les reliefs, on utilise 24 bits. Ce qui donne (24 * 14000 * 9600 * 260 / 8) octets, soit plus de 97 Gigaoctets en Bitmap. Mais les images sont distribuées en format TIFF, autre format très connu, qui permet de compresser des images avec plusieurs algorithmes. Et dans le format TIFF-LZW, les images font 4 gigaoctets pour toute la Suisse (gain de 95% !).

Un autre site parlait des méthodes d'améliorations de la compression LZW. Il proposait de créer un second tableau où seraient enregistrés les tailles des chaînes. C'est à peu près le même moyen que j'avais trouvé. Sauf que lui indique que le gain est de 1200 %, alors que pour moi il n'a été que de 500%. Il précise aussi, naturellement, que pour aller plus vite, il suffit que le programme sache où chercher la chaîne (si elle existe) dans le dictionnaire, sans a avoir a comparer toutes les chaînes (ce que j'avais bien compris). Pour cela, "*Il ne faut pas que deux chaînes différentes puissent avoir le même emplacement dans le dictionnaire. Ceci oblige donc à une gestion très poussée de la mémoire, qui oblige à en réserver de grandes quantités.*" Et malheureusement une *gestion très poussée* de la mémoire n'entre pas dans mes capacités.