# TP Compression Image

BEULE DAUZAT Rémy et TONNEAU Florent

Novembre 2015

## 1 Formulation du problème en programmation dynamique

### 1.1 Définition des variables

Objectif : Minimiser le nombre de bits

- c(i, n, b) : coût en nombre de bits entre i et $m^2$ pixels, pour une taille de segment n et un nombre de bit par pixel b
- $m^2$ : nombre de pixels dans le fichier
- i : pixel numéro i
- ai : nombre de bits du pixel i
- b : nombre de bits d'un pixel dans la séquence courante
- n : taille du segment courant

### 1.2 Algorithme récursif

$c(i, n, b) =$

- Si la séquence est rempli ou qu'il n'y en a pas (si $n >= 255$ ou $n = 0$) :
  retourne $11 + ai + c(i + 1, 1, ai)$
- Si le pixel est parfait pour la séquence ($ai = b$) :
  retourne $b + c(i + 1, n + 1, b)$
- Si le pixel est plus petit ($ai < b$) :
  retourne MIN :
  - $b + c(i + 1, n + 1, b)$ : on choisit de le garder quand même
  - $11 + ai + c(i + 1, 1, ai)$ : on ferme la séquence et on en ouvre une nouvelle
- Si le pixel est plus grand ($ai > b$) :
  retourne MIN :
  - $n * (ai - b) + ai + c(i + 1, n + 1, ai)$ : on modifie la séquence courante pour y rentrer le pixel i
  - $11 + ai + c(i + 1, 1, ai)$ : on ferme la séquence et on en ouvre une nouvelle

Condition d'arrêt : $i = m^2 + 1$ retourne $c(i) = 0$.

On démarre l'algorithme avec c(0,0,0).

### 1.3 Algorithme itératif

$c(i, n, b) =$

Pour tous les pixels en partant du dernier pixel de l'image :

- Pour tous les combinaisons (n, b) du pixel calculé à l'itération précédente :
  - Si la séquence est rempli ou qu'il n'y en a pas (si $n >= 255$ ou $n = 0$) :
    stocke la combinaison (1, ai) = $11 + ai$+combinaison (n, b)
  - Si le pixel est parfait pour la séquence ($ai = b$) :
    stocke la combinaison (n + 1, b) = $b$+ combinaison (n, b)
  - Si le pixel est plus petit ($ai < b$) :

- stocke la combinaison (n + 1, b) = $b$+ combinaison (n,b) : on choisit de le garder quand même
- stocke la combinaison (1, ai) = $11 + ai$+ combinaison (n,b) : on ferme la séquence et on en ouvre une nouvelle

— Si le pixel est plus grand ($ai > b$) :
- stocke la combinaison (n + 1, ai) = $n * (ai - b) + ai$+ combinaison (n,b) : on modifie la séquence courante pour y rentrer le pixel i
- stocke la combinaison (1, ai) = $11 + ai$+ combinaison (n,b) : on ferme la séquence et on en ouvre une nouvelle

On démarre l'algorithme avec c(0,0,0).

### 1.4 Tableaux de mémorisation

Pour la mémorisation des combinaisons, nous gardons des dictionnaires dans chacune des cases d'un tableau de $m^2$ cases. Chaque dictionnaire comporte les combinaisons (n, b) testées à partir du pixel courant i.
Par exemple, si nous avons une image de 16 pixels : la ième case (comprise entre 0 et 16) possède un dictionnaire qui correspond aux combinaisons possibles de (n, b) pour ce pixel donné. Les combinaisons permettent de savoir où se situe le pixel i dans une séquence de n pixels où tous les pixels sont codés sur b bits.
Aussi nous gardons dans un second tableau, tous les meilleurs coûts à partir de la case i. Par exemple, si nous prenons la case i de ce second tableau, la valeur représentera le meilleur coût en nombre de bits pour compresser les pixels de i à $m^2$.

### 1.5 Déterminer le chemin vers la meilleure solution

Pour retrouver la meilleur séquence de choix à partir du tableau des combinaisons, nous avons mis en place un algorithme qui itère sur le tableau de mémoisation :

On part du meilleur choix et donc de la racine de la meilleure combinaison du pixel 0.
Pour toutes les pixels du début à la fin du tableau :

— Si le n du pixel précédent = 1 : alors on cherche la combinaison pour laquelle son coût est égale au meilleur coût précédent moins le côut d'une entête et le coût en bits du pixel précédent. Ce cas correspond enfaite à l'ouverture d'une nouvelle séquence lors du calcul du meilleur coût.

— Sinon, si le n du pixel précédent = $n - 1$ et que le b du pixel précédent = b : alors cette combinaison peut appartenir à la séquence que l'on suit (la meilleure) car elle possède des paramètres cohérent par rapport à ceux du pixel précédent.
Si le b est inférieur strictement au b précédent alors cela veut dire que il y a eu un refactoring dans la séquence courante entre le pixel précédent et le pixel i.
On prend la combinaison qui possède le meilleur coût afin de rester sur la meilleur séquence.

De cette manière, on peut savoir si la meilleur solution fait le choix d'ouvrir une nouvelle séquence au pixel i ou de garder le pixel dans la séquence courante ou même encore de refactorer la séquence grâce au tableaux de mémorisation mis en place dans notre algorithme itératif. A chacun de nos choix, on est sûr qu'une solution existe pour suivre la meilleure séquence car si une case du tableau de mémoisation existe, cela signifie qu'une séquence permet d'en partir et d'arriver au bout du tableau.

### 1.6 Décompression d'un fichier

Pour décompresser un fichier, nous récupérons un tableau d'octets extrait du fichier compressé. On parcourt ensuite tous les octets de ce tableau.

— Si $n = 0$, alors on arrive sur l'entête d'une nouvelle séquence. Il faut récupérer la taille de cette séquence et le nombre de bits significatifs de chaque pixel. Au démarrage, cela est simple car la

taille de la séquence est codée sur le premier octet et b sur les trois premiers bits significatifs du deuxième. Pour la suite, on applique le même traitement que lors du démélage des pixels.
— Sinon, on est entrain de parcourir une séquence, b et n sont déjà fixés. Il faut donc juste dissocier les pixels entre eux, même s'il sont sur plusieurs octets. On utilise donc une variable stockant le nombre de bits encore non utilisés dans l'octet en cours. Il y a alors trois possibilités : soit le pixel est codé sur ce qu'il reste de l'octet ; soit il est dans l'octet et il ne prend pas tous les bits restants ; soit il déborde et il faut donc aller chercher son reste sur l'octet suivant.

# 2 Complexité des algorithmes

## 2.1 Complexité de l'algorithme récursif

L'algorithme récursif parcours toutes les combinaisons de (i, n, b) avec un i compris entre 0 et $m^2$, un n compris entre 1 et 255 et un b entre 1 et 8. Donc dans le pire des cas, nous devrions avoir $m^2 * 8 * 255$ cas, ce qui donne un algorithme linéaire par rapport au nombre de pixels de l'image.
$\Theta(\mathrm{m}^2)$ avec $m^2$ : nombre de pixels de l'image

## 2.2 Complexité de l'algorithme itératif

Dans l'algorithme itératif, nous itérons sur tous les pixels puis pour chacun de ces pixels, sur toutes les combinaisons (b, n) pour le i précédent. Sachant qu'un pixel i donné comporte au maximum $255 * 8$ combinaisons alors on devrait arriver à la même complexité que l'algorithme itératif. Nous avons donc un coût linéaire par rapport au nombre de pixels.
$\Theta(\mathrm{m}^2)$ avec $m^2$ : nombre de pixels de l'image

## 2.3 Complexité de la mémorisation

Rappellons que notre mémorisation comporte deux tableaux de $m^2$ cases, dont un tableau qui comporte toutes les combinaisons (n, b) testées au pixel i et le deuxième tableau stocke des entiers.

— tableau des combinaisons = $m^2 * 8 * 255$ combinaisons dans le pire des cas.
— tableau des meilleurs coûts = $m^2$

En réalité, on n'arrive que très rarement dans le pire cas, car à chaque itération, n ne peut évoluer que de 2 façon différente (avancer d'un dans la séquence ou redémarrer une séquence). De même pour b qui peut soit rester le même, soit être remplacé par le nombre de bits significatifs du pixel i (ai). Seul les réfactors augmentent le nombre de possibilités de b différents, mais ceux ci ne sont intéressants que dans les 11 premiers pixels d'une séquence(dans le meilleur des cas celui ou la différence entre b et ai égale 1). Après, il devient plus intéressant de créer une nouvelle séquence.

Dans la pratique, nous avons en moyenne 255 combinaisons dans chaque case pour un pixel donné car les combinaisons ont souvent le même b du fait que les pixels proches ont aussi des valeurs très proches et donc il y a très peu de changements du nombre de bits significatifs.

# 3 Implémentation et Tests

## 3.1 Récursif

Nous avons implémenté en premier lieu notre algorithme récursif en Python mais nous n'avons pas pu le tester au vu du nombre de récursion qui était trop important pour ce langage. Ensuite, nous avons décidé de le mettre en place en Java. Cependant après 3h d'éxécution il ne donne pas de résultat car trop lourd en mémoire pour la JVM malgré l'augmentation de sa taille dans les paramètres ($-Xss1024m - Xmx8196m$). Celle-ci a besoin de beaucoup de mémoire et elle a besoin de swapper

les données de l'algorithme.
Au final, nous n'avons jamais réussi à faire fonctionner cette algorithme.

### 3.2 Itératif

Pour l'algorithme itératif, nous l'avons implémenté en Python et comme son homologue, il prend énormément de place en mémoire cependant celui-ci arrive à finir en 2 minutes sur une machine comportant 8Go de mémoire vive.

### 3.3 Solution

N'ayant qu'un algorithme itératif qui fonctionne en Python, nous avons implémenté l'algorithme de recherche de la solution du meilleur coût en Python et nous arrivons à trouver la solution à partir des tableaux de mémorisation.

### 3.4 Questionnements sur notre implémentation

A vrai dire à ce jour, nous ne comprenons pas pourquoi nos algorithmes prennent autant de place en mémoire sachant que nos tableaux de mémorisation devrait prendre environ 200Mo pour des fichiers de 256ko. Ce pourrait-il que nous ayons des fuites mémoires ?