Algorithmique avancée, TP compression d'image

Nathanael COURET, Thomas BIANCHINI

Novembre 2015

1 Equation de récurrence

L'algorithme de compression est défini par l'équation de récurrence suivante: Soit S(i) le coût en bit de la meilleure séquence démarrant au pixel i, j le nombre de pixels dans la séquence courante, n le nombre total de pixels et c(i) la compression du pixel i.

$$S(i) = min_{0<j<256/i<n}(11 + j * c(i) + S(i + j + 1)) \text{ avec } S(n + 1) = 0$$

Pour chaque pixel, nous allons tout d'abord calculer la meilleur sequence de son suivant jusqu'à arriver à la fin de l'image. Nous calculons ici la meilleure facon d'arranger le dernier pixel, dans ce cas il s'agira simplement de créer une séquence d'un seul pixel et de taille 11+1*(8-c(n)). Ceci nous donne donc la S(n). Puis nous remontons à S(n-1) qui lui peut se décomposer en deux séquences d'un pixel ou une séquence de deux pixels. Il faut donc calculer 11+1*(8-c(n-1)) + S(n) (avec S(n) connue à ce stade) puis vérifier que l'on peut compresser P(n) et P(n-1) avec la même compression et calculer 11+2*(8-c(n,n-1)). Enfin nous prenons pour S(n-1) le minimun entre ces deux possibilités. Ainsi nous connaissons désormais S(n-1) et S(n) qui sont optimales. Nous faisons ceci jusqu'à S(0) en s'assurant que nous n'ajoutons pas plus de 256 pixels dans chaque séquence.

En se basant sur cette formule de récurrence, soit T(n) le nombre d'opérations faites durant l'algorithme, T(n) = 0(n). En effet dans le pire des cas nous devons calculer pour tous les pixels 256 possibilités ce qui nous donne une complexité de 256*n.

2 Implémentation

2.1 Lecture et écriture

- **lecture raw**: lecture ligne par ligne, alternant le sens à chaque fois pour la lecture en zigzag. Tous les octets sont enregistrés dans un tableau de taille n.

- **écriture seg**: Chaque segment calculé est enregistré avec la taille du segment sur 8 bits, puis le codage sur 3 bits et enfin les pixels encodés. Afin de pallier au problème de l'écriture de groupes de moins de 8 bits, nous avons utilisé la biliothèque bit-io qui permet d'écrire et lire un nombre arbitraire de bits. Il faut cependant rajouter un padding pour compléter le dernier octet du fichier.

- **lecture seg**: Comme pour l'écriture, la bibliothèque it-io est utilisée. chaque séquence est lue et les pixels récupérés sont enregistrés dans un tableau de byte.

- **écriture raw**: Les octets récupérés lors de la lecture seg sont simplement écrits dans le fichier cible, ligne par ligne en zigzag pour récupérer un fichier identique à l'original.

2.2 Implémentation de l'algorithme

Nous avons choisi d'utilier le langage de programmation Java car il offre une taille de pile modulable au besoin et de taille importante de base.
Voici la méthode récursive de l'algorithme avec mémoisation et calcul du chemin des meilleurs séquences à partir du pixel i.

```
private int sequence(int idxPixel, int[] mem, Sequence[] chemin) {
    if (idxPixel == intputData.size()) {
        //arrivee en bout de tableau
        return 0;
    } else {
        //memoisation
        if (mem[idxPixel] == -1) {
            //permet d'etre sur de toujours stocker une valeur
            mem[idxPixel] = Integer.MAX_VALUE;

            //compression du pixel courant (nombre de 0 a enlever)
            int comp = intputData.get(idxPixel).getMaxCompression();
            for (int i = 0; i < 255; ++i) {
                //formule de recurrence
                int t = (i + 1) * (8 - comp) + 11 + sequence(idxPixel + i
                    + 1, mem, chemin);

                //stockage de la chaine minimum
                if (t < mem[idxPixel]) {
                    mem[idxPixel] = t;
                    Sequence n = new Sequence();
                    n.nbPixels = i + 1;
                    n.compression = comp;
                    chemin[idxPixel] = n;
                }
                //si possible de travailler avec le suivant
                if (idxPixel + i + 1 < intputData.size()) {
                    int newComp = intputData.get(idxPixel + i +
                        1).getMaxCompression();
                    if (comp > newComp)
                        comp = newComp;
                } else {break;} //inutile de faire toute la boucle
            }
        }
        return mem[idxPixel];
    }
}
```

et voici la méthode itérative :

```
private int sequenceIteratif(int[] mem, Sequence[] sequence) {
       //calculer du "cas de base" en partant de la fin
       for (int idxPixel = intputData.size() - 1; idxPixel >= 0;
           idxPixel--) {
          int d = intputData.get(idxPixel).getMaxCompression();
          for (int j = 0; j < 255; ++j) {
              //initialisation de n+1 a 0
              int t = 11 + (j + 1) * (8 - d) + mem[idxPixel + j + 1];
              if (t < mem[idxPixel]) {
                  mem[idxPixel] = t;
                  Sequence n = new Sequence();
                  n.nbPixels = j+1;
                  n.compression = d;
                  sequence[idxPixel] = n;
              }
              if (idxPixel + j + 1 < intputData.size()) {
                  if (d > intputData.get(idxPixel + j +
                      1).getMaxCompression())
                     d = intputData.get(idxPixel + j +
                         1).getMaxCompression();
              } else
                  break;
          }
       }
       return mem[0];
   }
```

Afin de parcourir le chemin il suffira d'accéder à la séquence stocké en i, regarder le nombre de pixel et accéder à l'indice [nombre de pixel] du tableau de séquence pour accèder à la suite. Voici la méthode :

```
private void displayChemin(Sequence[] chemin, int begin) {
    while (begin < intputData.size()){
        System.out.println(
               "Sequence [pixels: " + chemin[begin].nbPixels + ", comp:
                   " + chemin[begin].compression + "] ");
        begin += chemin[begin].nbPixels;
    }
}
```

2.3 Tests

Afin d'avoir une démarche évitant la régression nous avons utilisé les tests unitaires sur des exemples "simplesé à calculer à la main puis nous avons utilisé les images fournies afin de calculer les performances de nos algorithmes. Cette partie sera décrite après.

3 Performances

3.1 Compression

nous avons décidé d'étudier le taux de compression des images après leur traitement par notre algorithme. Le résulat pour les images de 512 pixels nous donne ceci :

Nous gagnons en moyenne 6% par rapport aux images d'origine. Or les images faisaient 512*512 = 262144 pour ne faire au final plus que 245000 pixels. Les gains paraissent négligeable mais cela est surement du au fait qu'une image toute noire se compressera beaucoup mieux qu'une image toute blanche pour ce type d'algorithme. D'autre part nous remarquons que la compression est nettement plus efficace sur les mêmes images mais cette fois ci agrandies. En effet le graphique suivant résume les nouveaux taux de compression :

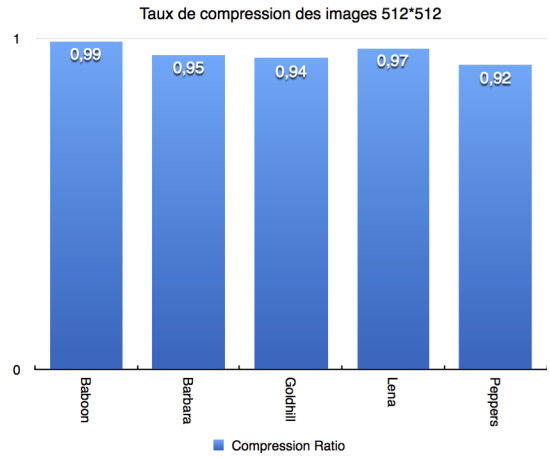

Figure 1: Ratio de compression des images de 512 pixels

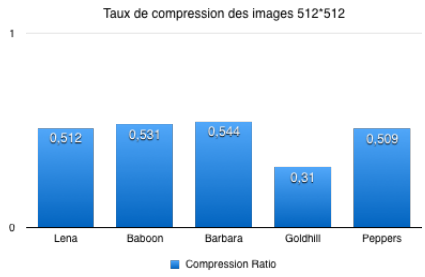

Figure 2: Ratio de compression des images de 1024 pixels

3.2 Temps

Après avoir réalisé quelques tests nous nous sommes rendus compte que le temps d'exécution de l'aglorithme itératif était nettement plus faible que celui de l'algorithme récursif. Cela s'explique par le coût mémoire de l'algorithme récursif utilisant une taille de pile très importante et ralentissant ainsi les calculs pour les images de taille conéquente. En effet, en nombre d'opération les deux algorithmes se valent. Cependant le coût mémoire de la récursivité est ressentie. Voici un graphique comparant les temps d'exécution :

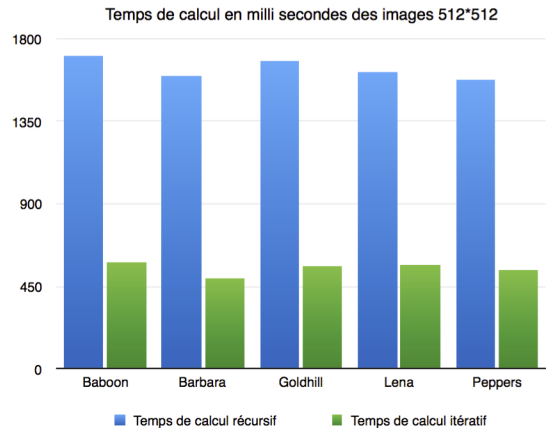

Figure 3: Comparaison des temps d'exécution

Nous nous penchons ici seulement sur le nombre d'opérations effectuées par l'algorithme. Ainsi on remarque que le temps de calcul est proportionnel par rapport aux images de 512 avec des images de 1024 (4 fois plus grand dû à notre complexité en O(n). Le graphique suivant nous montre ce résulat:

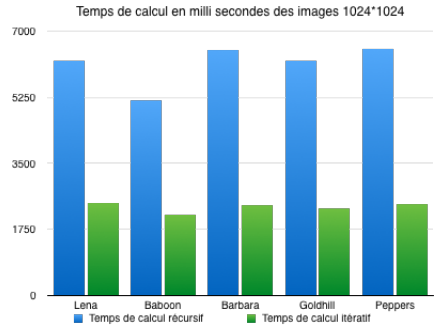

Figure 4: Comparaison des temps d'exécution

4 Conclusion

Pour conclure ce TP nous a fait prendre conscience que peu de langages “modernes” manipulent des données plus petites que l’octet et qu’il faut descendre en niveau de langage pour pouvoir travailler des images bits à bits.
De plus et certainement le plus important, nous avons pu appliquer la programmation dynamique sur un problème de compression (concret) et voir que cette technique de programmtion améliore grandement l’efficacité du traitement (en opposition à une soltion naïve d’énumération de toutes les combinaisons possibles sans mémoïsation).
Ce TP nous a aussi montré que le coût mémoire d’un algorithme doit être considéré. En effet les deux algorithmes réalisent le même nombre d’opérations mais la récursivité augmente drastiquement le temps d’exéctuion du programme à cause de son coût mémoire.