Université de Liège

Projet 3 - Analyse d'images

Structure de données et algorithmes

Maxime Meurisse
Valentin Vermeylen

2e année de Bachelier Ingénieur Civil
Année académique 2017-2018

1 Résolution analytique

L'erreur associée à la compression de l'image est définie par

$$Err\left(g\right)=\sum_{i=0}^{n-1}h\left[i\right]\left(i-g\left(i\right)\right)^{2} \tag{1}$$

où $h\left[i\right]$, pour $i=0,...,n-1$, est le nombre de pixels de l'image de valeur i.

Le problème consiste à partitionner la fonction g de la sorte :

$$g\left(i\right)=\begin{cases} v_1 & \text{si} \quad 0\leq i<p_1 \\ v_2 & \text{si} \quad p_1\leq i<p_2 \\ ... & \\ v_k & \text{si} \quad p_{k-1}\leq i<n \end{cases}$$

Le but est de minimiser (1). En dérivant par rapport à g et en égalant cette dérivée à 0, on obtient

$$\frac{\mathrm{d}Err\left(g\right)}{\mathrm{d}g}=\sum_{i=0}^{n-1}-2h\left[i\right]\left(i-g\left(i\right)\right)=0 \tag{2}$$

L'expression (2) peut se ré-écrire en tenant compte de la partition en k intervalles :

$$\sum_{j=1}^{k}\sum_{i=p_{j-1}}^{p_j-1}-2h\left[i\right]\left(i-g\left(i\right)\right)=0 \tag{3}$$

Or on sait que dans chaque intervalle (pour chaque valeur de k), la fonction g aura une valeur v fixée. On peut donc ré-écrire l'expression (3) sous la forme

$$\sum_{i=p_{j-1}}^{p_j-1}-2h\left[i\right]\left(i-v_k\right)=0$$

Et en isolant v_k, on obtient finalement

$$v_k=\frac{\sum_{i=p_{j-1}}^{p_j-1}h\left[i\right]i}{\sum_{i=p_{j-1}}^{p_j-1}h\left[i\right]} \tag{4}$$

Cette expression analytique peut être évaluée en un temps linéaire par rapport à la taille de l'intervalle $[p_{j-1},p_j[$ en profitant de la formule d'erreur. Le problème algorithmique consiste donc bien à déterminer uniquement les valeurs des p_j.

2 Approche par recherche exhaustive

Soit l'histogramme h de taille n. Le but est de fractionner cet histogramme en intervalles. Il faut donc déterminer $k-1$ valeurs de p. Ces valeurs de p sont des positions d'éléments de l'histogramme, qui serviront de bornes aux intervalles.

La première position de l'histogramme, 0, ne peut être choisie car toutes les valeurs de p sont strictement supérieures à 0. Il reste donc $n-1$ positions possibles pour placer les valeurs de p.

Le nombre de façons de placer $k-1$ valeurs sur $n-1$ positions possibles est donné par

$$C_{n-1}^{k-1} = \frac{(n-1)!}{(n-k)!\,(k-1)!}$$

Dans tous les cas, il faut placer $k-1$ valeurs de p. La complexité est donc donnée par $\Theta\left(C_{n-1}^{k-1}\right)$, à savoir une complexité exponentielle.

3 Approche gloutonne

3.a Solution au problème

La solution par approche gloutonne est une solution itérative qui travaille dans des intervalles différents à chaque itération. L'intervalle de base est l'histogramme tout entier.

À chaque itération :

1. on prend la position de l'élément du milieu de l'intervalle comme seuil ;
2. on calcule les erreurs des sous-intervalles à gauche et à droite de ce seuil ;
3. on stocke les intervalles et leurs erreurs associées dans un tableau ;
4. on cherche dans ce même tableau l'intervalle ayant la plus grande erreur ;
5. on recommence les opérations dans cet intervalle.

L'algorithme procède à $k-1$ itérations, avec k, le nombre de niveaux désirés pour la compression. Un intervalle de taille 1 ayant une erreur de 0, on est sûr que la séparation en k intervalles pourra toujours se faire pour autant que $k < n$, avec n, le niveau maximal des pixels.

Lorsque tous les p_i sont déterminés, les v_i sont calculés sur base de la relation (4).

3.b Complexité de la solution

La fonction `computeReduction` contient deux boucles imbriquées. La première boucle, dans tous les cas, fera $k-1$ itérations. En effet, il faut déterminer $k-1$ seuils. La deuxième boucle, amenée par la fonction `computeError`, fera $\frac{n}{2}$ itérations dans le pire cas. Elle devra en effet calculer l'erreur sur les deux moitiés de l'histogramme de taille n pour l'intervalle de base, à savoir tout l'histogramme. Les autres fonctions présentes dans l'algorithme ont une complexité moindre et ne bornent donc pas la complexité de celui-ci.

Dans tous les cas, la complexité sera donc donnée par $\Theta\left(nk\right)$.

3.c Contre-exemple

Cette approche ne fournit pas la solution optimale dans tous les cas.

Soit un histogramme de taille 4 définit par $h = [0, 0, 1, 7]$. Si cette l'image correspondante est compressée sur 2 niveaux ($k = 2$), l'algorithme placera l'unique valeur de p au milieu, à savoir $p = 2$.

Dans ce cas de figure, la fonction g est donnée par

$$g(i) = \begin{cases} v_1 & \text{si} \quad 0 \leq i < 2 \\ v_2 & \text{si} \quad 2 \leq i < 4 \end{cases}$$

et on constate aisément que l'erreur sera non-nulle.

Si on avait eu $p = 3$, la fonction g aurait été donnée par

$$g(i) = \begin{cases} v_1 & \text{si} \quad 0 \leq i < 3 \\ v_2 & \text{si} \quad 3 \leq i < 4 \end{cases}$$

et l'erreur aurait été nulle en donnant à v_1 la valeur 1 et à v_2 la valeur 7. Dans ce cas-ci, cette solution est donc plus optimale que celle obtenue via l'algorithme glouton.

4 Approche par programmation dynamique

4.a Fonction de coût

La formulation récursive complète est donnée par

$$ErrMin(n,k) = \begin{cases} err(0,n) & \text{si} \quad k = 0 \\ \underset{k-1<i\leq n-1}{min}\{err(i,n) + errMin(i,k-1)\} & \text{si} \quad k \in [1, n-1] \\ 0 & \text{si} \quad k = n \end{cases}$$

où ERR(I, J) est l'erreur calculée sur l'intervalle [i, j[.

4.b Graphe des appels récursifs

Le graphe des appels récursifs pour des couples de valeurs (n, k) est donné à la figure 1. On constate sur ce graphe que des cas similaires sont calculés plusieurs fois.

4.c Pseudo-code

Le pseudo-code d'un algorithme efficace de calcul des valeurs est donné ci-dessous.

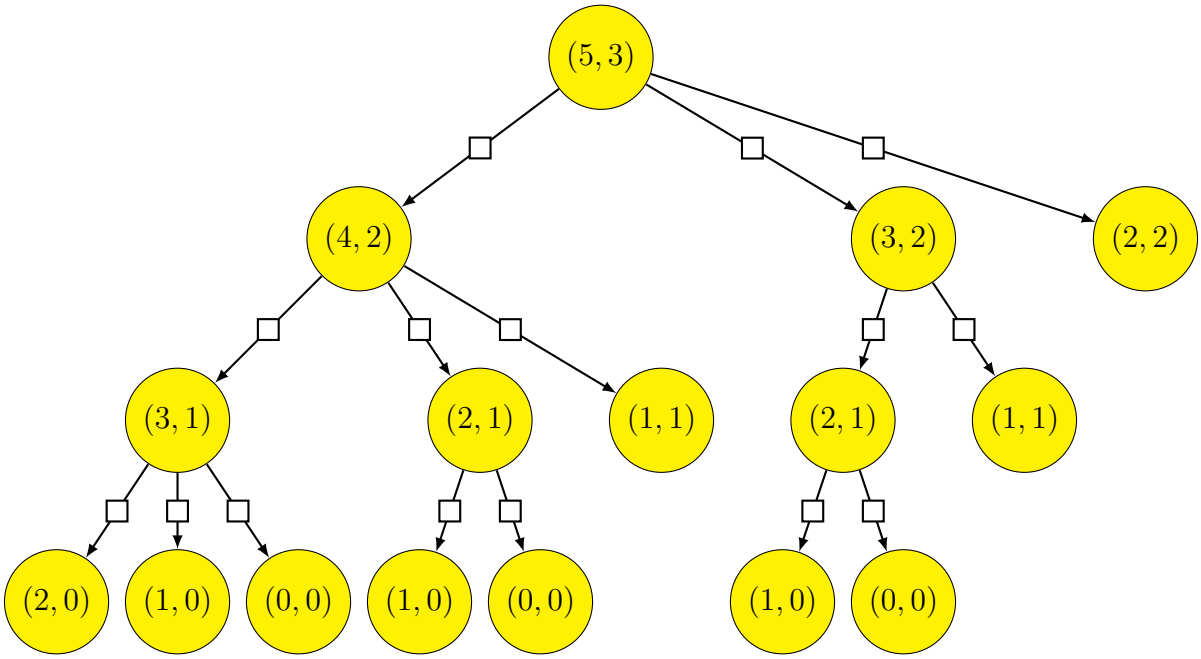

Figure 1 – Graphe des appels récursifs pour des couples de valeurs (n, k).

```
1  errMin(n, k, error, intervals, histogram, nLevels) {
2      err = 0
3      minErr = +INFINITY
4      index = getIndex(n, k, nLevels)
5
6      if(k == n)
7          error[index] = 0;
8          return 0;
9
10     if(k == 0)
11         error[index] = computeError(histogram, 0, n);
12         return error[index];
13
14     if(error[index] != +INFINITY)
15         return error[index];
16
17     for(i = n - 1; i > k - 1; i--)
18         err = computeError(histogram, i, n) + errMin(i, k - 1, error,
               intervals, histogram, nLevels);
19
20         if(err < minErr)
21             minErr = err;
22
23     error[index] = minErr;
24
25     return minErr;
26 }
```

Listing 1 – Pseudo-code de l'approche par programmation dynamique.

4.d Solution optimale

Le pseudo-code précédent a été modifié afin de récupérer la solution optimale.

```
 1 errMin(n, k, error, intervals, histogram, nLevels) {
 2     err = 0
 3     minErr = +INFINITY
 4     index = getIndex(n, k, nLevels)
 5
 6     if(k == n)
 7         error[index] = 0;
 8         return 0;
 9
10     if(k == 0)
11         error[index] = computeError(histogram, 0, n);
12         intervals[index] = n;
13         return error[index];
14
15     if(error[index] != +INFINITY)
16         return error[index];
17
18     for(i = n - 1; i > k - 1; i--)
19         err = computeError(histogram, i, n) + errMin(i, k - 1, error,
               intervals, histogram, nLevels);
20
21         if(err < minErr)
22             minErr = err;
23             intervals[index] = n - i + 1;
24
25     error[index] = minErr;
26
27     return minErr;
28 }
```

Listing 2 – Pseudo-code de l'approche par programmation dynamique modifié pour récupérer la solution optimale.

4.e Complexité en temps et en espace

Dans le ComputeReduction, une bonne partie des instructions sont O(1). Les fonctions ComputeLevel et getIndex possède des complexités constantes et ComputeError est O(n). On se rend donc vite compte que l'instruction dont la complexité va régir la complexité du code est l'appel récursif à errMin avec les arguments *n-1* et *k-1*. Ce premier appel va lui engendrer $n-k$ appels récursifs et ainsi de suite. On se rend rapidement compte que la complexité d'un tel algorithme récursif est très difficilement calculable. Néanmoins, en exprimant le problème de manière itérative, on peut montrer que la complexité est $O(n^3)$. En effet, il suffit de remplir le tableau des erreurs (de taille n x n) en commençant par la ligne $k = 0$ et en remontant les valeurs de k, ce qui se fait en un temps au cube. Le remplissage se fait en un temps quadratique, mais la détermination de chaque erreur se fait en un temps linéaire (il faut considérer au pire cas la totalité des cases de la ligne

supérieure, si on recherche l'erreur de n,1). Nous avons donc une complexité $O(n^3)$. Il est également à noter que $\theta(n^3) + \Theta(n^2 k)$ est égal à $\Theta(n^3)$ étant donné que k $\leq$ n.

5 Implémentation

5.a Approche gloutonne

Comme mentionné dans la section 3.a, les erreurs calculées pour chaque sous-intervalle sont sauvegardées. Ce choix a été fait dans le but de pouvoir tenir compte de tous les sous-intervalles et non se restreindre à ceux de l'intervalle étudié.

Les erreurs sont sauvegardées dans un tableau dont chaque élément est une structure contenant le début et la fin de l'intervalle, ainsi que son erreur.

Les sous-intervalles étant étudiés dans un ordre aléatoire, les seuils déterminés ne sont pas ordonnés par ordre croissant. Afin de corriger cela, le tableau contenant ces seuils est trié avec l'algorithme `InsertionSort` lorsque tous les seuils ont été déterminés.

6 Analyse empirique

Les résultats fournis au tableau 1 sont des moyennes de 10 essais, tous réalisés sur des histogrammes de tailles croissantes générés aléatoirement, avec un nombre de niveaux après compression égal à 10.

n	`GreedyReduction`	`DPReduction`
10	0,000 011	0,000 009
100	0,000 020	0,007 973
1000	0,000 049	6,222 055
10 000	0,000 198	x
100 000	0,001 455	x
1 000 000	0,015 384	x

Tableau 1 – Temps d'exécution, en secondes, des différents algorithmes de compression d'image pour différentes tailles n d'histogrammes générés aléatoirement.

La complexité théorique de l'algorithme glouton, calculée au point 3.b, est $\Theta(nk)$. Dans les essais effectués, la valeur de k est fixée à 10. Lorsque l'on multiplie par 10 la taille de l'histogramme entre chaque nouveau essai, le nombre d'opérations à effectuer est lui aussi multiplier par 10. Il devrait en être de même pour les temps d'exécution. On constate dans la table 1 que c'est bien le cas.

Remarque Cette constatation est moins marquée pour de petites tailles d'histogramme. Cela vient certainement du fait que l'ordinateur possède un temps d'exécution minimum sous lequel il ne peut descendre.

Dans le cas de l'approche par programmation dynamique, le temps d'exécution augmente fortement avec la taille de l'histogramme. Ce résultat semble logique au vu de la complexité obtenues (le nombre d'opérations à faire est multiplié par 10 pour chaque nouvelle taille).

Pour des tailles d'histogramme supérieures à 1000, les temps n'ont pas pu être obtenus.

7 Comparaison des résultats et conclusion

7.a Comparaison visuelle

7.a.1 Compression sur 4 niveaux

Figure 2 – Compression sur 4 niveaux par l'approche naïve à gauche, par l'approche gloutonne au centre et par la programmation dynamique à droite.

Sur 4 niveaux, on se rend compte que l'approche dynamique offre plus de contrastes, là où l'approche naïve fournit un résultat plus marqué au moyen de tons de gris plus éloignés les uns des autres, ce qui est moins fidèle à l'image de base. L'approche gloutonne fournit un résultat assez similaire à l'approche par programmation dynamique, mais offre moins de contraste. L'approche par programmation dynamique semble fournir le résultat le plus fidèle à l'original et supprime moins de détails au niveau du visage que l'approche gloutonne.

7.a.2 Compression sur 10 niveaux

Sur 10 niveaux, la différence entre les approches reste marquée. L'approche naïve fournit un résultat assez grossier tandis que les deux autres donnent des images plus précises. Au

Figure 3 – Compression sur 10 niveaux par l'approche naïve à gauche, par l'approche gloutonne au centre et par la programmation dynamique à droite.

niveau de l'épaule et de l'objet sur la droite, on voit que l'approche DPR est plus précise que son homologue gloutonne.

7.a.3 Compression sur 50 niveaux

Figure 4 – Compression sur 50 niveaux par l'approche naïve à gauche, par l'approche gloutonne au centre et par la programmation dynamique à droite.

Sur 50 niveaux, les différences s'amenuisent. Les tons du visage sont peut-être moins marqués sur l'image naïve mais les deux autres sont fortement identiques. Pour de faibles valeurs de k, il est plus intéressant d'utiliser l'approche dynamique mais quand k s'approche de 50, l'approche gloutonne fournit un résultat aussi satisfaisant.