Structure de données et algorithmes

Projet 3 : Mise en page automatique d'une bande dessinée

LEWIN Sacha

MAKEDONSKY Aliocha

Ingénieur civil 2^{e} bachelier
Année académique 2019-2020

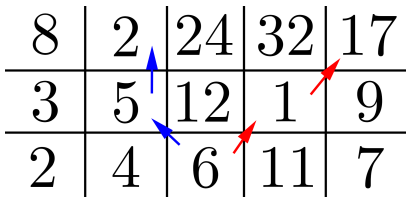

FIGURE 1 – Contre-exemple qui démontre la non-optimalité

1 Complexité d'une approche exhaustive

Soit une image de taille $m \times n$ Dans le calcul d'une recherche exhaustive, on répète k fois les recherche et suppression d'un sillon d'énergie minimale. La recherche se fait en comparant tous les sillons se terminant sur chacune des lignes. On recherche et compare donc, pour chacun des m pixels de la dernière ligne, tous les sillons qui terminent sur ceux-ci. Pour chacun de ces pixels, la recherche consiste à remonter vers le haut. On part d'un pixel, il y a 3 pixels voisins à celui-ci sur la ligne au-dessus, pour chacun de ces 3 pixels, on appelle la fonction de coût. Ainsi, pour chacun des 3, on 3 nouveaux choix pour continuer le chemin, et ainsi de suite. On a donc 3^{n-1} chemins possibles, où n est la hauteur, ce qui est exponentiel.

On obtient au final une complexité de l'ordre de $\Theta(k \cdot m \cdot 3^n)$, où k est le nombre de pixels à retirer, m la largeur de l'image, et n la hauter de l'image.

2 Approche gloutonne non-optimale

Une approche gloutonne serait de procéder comme suit : On part du pixel (i, j) et on remonte. À chaque itération, on a (au maximum, selon si on est à côté d'un bord ou pas) trois choix (les 3 pixels voisins et situés sur la ligne du dessus). Sur ces trois choix, on choisirait alors à chacune de ces itérations le choix optimal local, c'est-à-dire le pixel d'énergie la plus faible. On effectue ceci jusqu'à arriver au sommet de l'image.

Cette approche, en plus d'être simple à implémenter, réduit considérablement la complexité de l'algorithme. En effet, il n'y a plus de complexité exponentielle, puisqu'on remonte simplement l'image de bas en haut en sélectionnant à chaque fois un pixel. Elle est donc maintenant linéaire par rapport à n : $\Theta(k \cdot m \cdot n)$.

Cependant, elle n'est en effet **pas optimale** ! Cela se démontre à l'aide d'un simple contre-exemple, comme celui à la Figure 1. Supposons une image de dimensions 5×3, et on cherche le sillon optimal s'arrêtant au pixel (3, 3). La grille représente les 15 pixels, et pour chacun de ceux-ci est marquée son énergie respective. En rouge est représenté le chemin que l'algorithme glouton suivrait, et en bleu le réel sillon optimal. Comme on peut le voir, le sillon optimal possède une énergie totale de $6+5+2 = 13$ alors que la solution gloutonne nous fournit un sillon d'énergie totale $6 + 1 + 17 = 24$.

3 Formulation mathématique de la fonction de coût

Considérons toujours une image de taille $m \times n$. Nous pouvons formuler récursivement notre fonction de coût (c'est-à-dire d'énergie minimale de sillon) de la sorte :

$$C(i,j) = E(i,j) + \begin{cases} 0 & \text{si } i = 1 \\ \min(C(i-1,j), C(i-1,j+1)) & \text{Si } i > 1 \text{ et } j = 1 \\ \min(C(i-1,j-1), C(i-1,j)) & \text{Si } i > 1 \text{ et } j = m \\ \min(C(i-1,j-1), C(i-1,j), C(i-1,j+1)) & \text{Si } i > 1 \text{ et } 1 < j < m \end{cases}$$

Où C(i, j) est l'énergie du sillon d'énergie minimal s'arrêtant en (i, j). Si nous sommes sur la première ligne, on ne prend simplement que l'énergie du pixel courant. Sinon, on y ajoute l'énergie du sillon d'énergie minimal qui est minimale parmi les 3 pixels voisins situés au-dessus. Il faut juste omettre le pixel voisin supérieur de gauche si on est sur le bord gauche, et le pixel voisin supérieur droit si on est sur le bord droit, puisqu'on sort de l'image.

4 Graphe des appels récursifs

Comme nous l'avons décrit par la formule mathématique récursive de al fonction de coût dans la section 3, le sillon d'énergie minimale comprenant le pixel de coordonnées (i, j) sera le sillon d'énergie minimale arrivant dans un des 3 pixels se situant au dessus de notre pixel (i, j), auquel on ajoutera bien sûr ce dernier. Par conséquent chaque pixel est lié aux à ses 3 voisins de la ligne juste au-dessus de lui, soit les 3 pixels de coordonnées $(i-1, j-1); (i-1, j); (i-1, j+1)$.

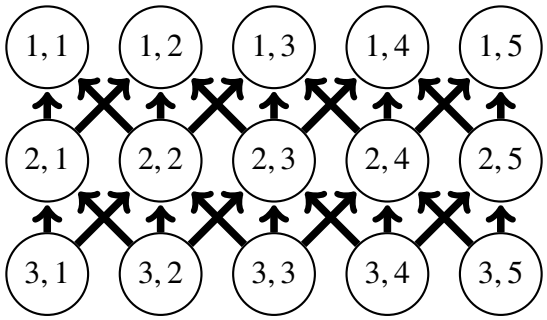

FIGURE 2 – Graphe des appels récursifs

5 Pseudo-code de l'implémentation efficace

```
1: function MIN_ENERGY_PATH(Energies)
2:     width = Energies.width
3:     height = Energies.height
4:     tab = "New array of double, of size height × width"
5:     for i = 1 to height do
6:         for j = 1 to width do
7:             tab[i, j] = Energies[i, j]
```

```
            if i > 1 then
                if j = 1 then
                    tab[i,j] = min(tab[i - 1, j], tab[i - 1, j + 1])
                else if j = width then
                    tab[i,j] = min(tab[i - 1, j - 1], tab[i - 1, j])
                else
                    tab[i,j] = min(tab[i - 1, j - 1], tab[i - 1, j], tab[i - 1, j + 1])
                end if
            end if
        end for
    end for
    min = tab[height, 1]
    for j = 2 to width do
        new = tab[height, j]
        if new < min then min = new
        end if
    end for
    return min
end function
```

6 Pseudo-code de la réduction d'image

Avant de passer au pseudocode de la fonction permettant de renvoyer l'image modifiée, il faut tout d'abord procéder à un modification du pseudocode de la section précédente afin qu'il renvoie un tableau contenant tout le chemin à suivre et pas uniquement le coût du sillon optimal.

```
function MIN_ENERGY_PATH(Energies)
    width = Energies.width
    height = Energies.height
    tab = "New array of size height × width"
    moves = "New array of size height × width"
    for i = 1 to height do
        for j = 1 to width do
            tab[i, j] = Energies[i, j]
            if i > 1 then
                if j = 1 then
                    if tab[i-1,j] < tab[i-1,j+1] then
                        tab[i,j] = tab[i-1,j]
                        moves[i,j] = "top_mid"
                    else
                        tab[i,j] = tab[i-1,j+1]
                        moves[i,j] = "top_left"
                    end if
                else if j = width then
                    if tab[i-1,j-1] < tab[i-1,j] then
                        tab[i,j] = tab[i - 1, j - 1]
                        moves[i,j] = "top_left"
```

```
                    else
                        tab[i,j] = tab[i - 1, j]
                        moves[i,j] = "top_mid"
                    end if
                else
                    if tab[i-1,j-1] < tab[i-1,j] and tab[i-1,j-1] < tab[i-1,j+1] then
                        moves[i,j] = "top_left"
                        tab[i,j] = tab[i-1,j-1]
                    else if tab[i-1,j] < tab[i-1,j-1] and tab[i-1,j] < tab[i-1,j+1] then
                        moves[i,j] = "top_mid"
                        tab[i,j] = tab[i-1,j]
                    else
                        moves[i,j] = "top_right"
                        tab[i,j] = tab[i-1,j+1]
                    end if
                end if
            end if
            tab[i,j] = tab[i,j] + Energies[i,j]
        end for
    end for
    min = tab[height, 1]
    optimal_end_pixel = 1
    for j = 2 to width do
        new = tab[height, j]
        if new < min then min = new optimal_end_pixel = j
        end if
    end for
    path = "New array of size height"
    column = optimal_end_pixel
    for j = 1 to height do
        row = height - j
        path[row] = column
        switch moves[row,column]
            case "top_left"
                column = column-1
                break
            case "top_right"
                column = column+1
                break
        end switch
    end for
    return path
end function
```

Dans ce pseudocode, k sera le nombre de pixels à enlever. La fonction pixel_energy calcule l'énergie d'un pixel en fonction de ses coordonnées. min_energy_path est la fonction décrite par le pseudocode juste au-dessus.

```
1:  function REDUCEIMAGEWIDTH(image, k)
2:      original = image
3:      height = image.height
4:      width = image.width
5:      energy_tab = "New array of size height × width"
6:      for i = 1 to height do
7:          for j = 1 to width do
8:              energy_tab[i,j] = PIXEL_ENERGY(original,i,j)
9:          end for
10:     end for
11:     for m = 1 to k do
12:         new_width = width-1
13:         new_image = CREATEPNM(width-1,height)
14:         new_energy_tab = "New array of size new_width × height"
15:         min_path = MIN_ENERGY_PATH(energy_tab, width, height)
16:         for i = 1 to height do
17:             x = 1
18:             path_col = min_path[i]
19:             for j = 1 to width do
20:                 if j = path_col then
21:                     continue
22:                 end if
23:                 new_energy_tab[i,x] = energy_tab[i,j]
24:                 x = x+1
25:             end for
26:         end for
27:         for i = 1 to height do
28:             path_col = min_path[i]
29:             if path_col ≤ new_width then
30:                 new_energy_tab[i,path_col] = PIXEL_ENERGY(new_image, i, path_col);
31:             end if
32:             if path_col > 1 then
33:                 new_energy_tab[i,path_col - 1]
34:                         = PIXEL_ENERGY(new_image, i, path_col - 1);
35:             end if
36:         end for
37:         original = new_image
38:         energy_tab = new_energy_tab
39:         width = width-1
40:     end for
41:     return original;
42: end function
```

7 Complexité de la solution

Considérons une image de hauteur n (nombre de lignes), et longueur m (nombre de colonnes).

Analysons d'abord la complexité en temps. Nous répétons k fois notre procédé qui retire un sillon. Pour trouver le sillon, nous remplissons dans une double boucle le tableau des énergies des sillons d'énergie minimale (et le tableau des directions à suivre). Ce tableau 2D est de taille $n \times m$ et le parcours donne donc une complexité pour cette étape de $\Theta(n \times m)$. Ensuite, nous avons une boucle qui compare toutes les valeurs de la dernière ligne, donc une étape de complexité $\Theta(m)$. Ensuite, on construit le chemin à partir des directions à suivre précédemment calculées. Ici, on parcourt la hauteur de l'image, donc nous avons au total $\Theta(n)$. La dernière étape consiste à supprimer le sillon de l'image. Pour cela, nous devons recopier l'image et nous avons une double boucle qui parcourt du $n \times (m-1)$.

Au final, globalement, seule la complexité du remplissage des coûts importe. En effet, les autres sont plus petites ou égales asymptotiquement et n'influencent donc pas le comportement asymptotique global de l'algorithme.
La complexité en temps de l'algorithme est dès lors :

$$\Theta(k \cdot m \cdot n)$$

Intéressons-nous désormais à la complexité en espace. Nous stockons une nouvelle image, et un tableau des énergies ce qui demande une complexité en espace de $\Theta(n \times m)$ pour chacun des deux. Ensuite, dans la fonction qui calcule les coûts, nous créons un autre tableau $n \times m$ qui contient les énergies des sillons d'énergie minimale s'arrêtant à chaque pixel de l'image, et un autre tableau des directions à suivre à partir de chaque pixel pour suivre le sillon optimal, de même dimension $n \times m$. Encore une fois, nous avons $\Theta(n \times m)$ pour les deux. On stocke ensuite un tableau de taille n pour les colonnes du sillon à chaque ligne, donc $\Theta(n)$.

Globalement, nous obtenons donc une complexité en espace de :

$$\Theta(n \cdot m)$$