INFO-2050 : Mise en page automatique d'une bande dessinée
Rapport

Maxime GOFFART
180521

Olivier JORIS
182113

2019 - 2020

Dans ce rapport, on considère que m est la largeur et n est la hauteur d'une image. Dans le pseudo-code des parties 3.4 et 3.5, nous avons retiré les vérifications des retours des différentes fonctions pour ne pas alourdir la lecture du pseudo-code.

3.1 Approche exhaustive pour déterminer le sillon optimal

Une approche exhaustive pour déterminer le sillon optimal serait d'envisager tous les sillons possibles arrivant à un pixel de la dernière ligne de l'image et de prendre celui avec le coût minimal en énergie. Pour un pixel fixé du sillon, on a 3 possibilités pour le prochain : celui directement au-dessus, celui au-dessus à gauche ou celui au-dessus à droite. Étant donné que la dernière ligne, comme toutes les autres, comporte m pixels, on a m potentiels pixels faisant partie du sillon optimal pour celle-ci. La complexité d'une telle approche est donc en $\Theta(m*3^n)$ et est donc exponentielle par rapport à n.

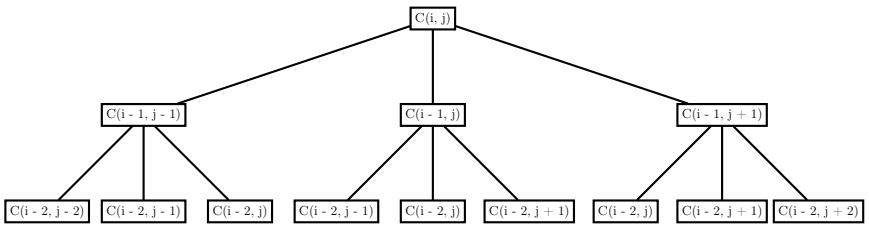
3.2 Formulation récursive du coût du sillon optimal

Soit C(i, j), le coût en énergie du sillon optimal s'arrêtant au pixel (i, j) avec $1 \leq i \leq n$ et $1 \leq j \leq m$.

$$\text{On a : C(i, j)=} \begin{cases} \text{E}(i,j) \ si\ i = 1 \\ \text{E}(i,j) + \min\{\text{C}(i-1,j-1),\ \text{C}(i-1,j)\}\ si\ j = m\ et\ i > 1 \\ \text{E}(i,j) + \min\{\text{C}(i-1,j),\ \text{C}(i-1,j+1)\}\ si\ j = 1\ et\ i > 1 \\ \text{E}(i,j) + \min\{\text{C}(i-1,j-1),\ \text{C}(i-1,j),\ \text{C}(i-1,j+1)\}\ sinon. \end{cases}$$

3.3 Graphe des appels récursifs

Le graphe ci-dessous représente les premiers appels récursifs de la fonction pour trouver le coût du sillon optimal s'arrêtant au pixel (i, j).



Si on continue la construction de ce graphe, celui-ci croît exponentiellement jusqu'à rencontrer un cas de base. Cette implémentation n'est pas optimale car les appels récursifs sont exponentiels et des résultats obtenus précédemment sont recalculés. Par exemple, on observe sur le graphe que le coût attribué au pixel [1] (i-2, j) est calculé 3 fois. Une approche par programmation dynamique faisant intervenir la mémoïsation de certains coûts réduirait le temps de calcul.

3.4 Pseudo-code du calcul du coût du sillon optimal

La fonction FIND_OPTIMAL_GROOVE prend en argument une table des coûts dont l'élément (i, j) renseigne sur le coût du pixel (i,j) et renvoie le sillon optimal.

1. Il s'agit d'un appel à C(i-2, j).

La fonction FIND_OPTIMAL_PIXEL prend en arguments la table des coûts, un indice de colonne et un indice de ligne. Cette fonction renvoie un pixel optimal à ajouter dans le sillon sur base du dernier pixel ajouté dans le sillon (dont les coordonnées sont passées en paramètres de la fonction).

```
FIND_OPTIMAL_GROOVE(cTable)
 1  optimalGroove = "Allocate a new Groove"
 2  optimalGroove.cost = 0
 3  optimalGroove.path = "Allocate a new path of size n"
 4  minLastLine = +∞
 5  positionLastLine = 0
 6
 7  for i = 1 to m // Trouver le pixel avec le plus petit coût sur la dernière ligne.
 8       if cTable[cTable.height][i] < minLastLine
 9            minLastLine = cTable[cTable.height][i]
10            positionLastLine = i
11  // Ajouter le pixel trouvé au sillon optimal.
12  optimalGroove.path[cTable.height].line = cTable.height
13  optimalGroove.path[cTable.height].column = positionLastLine
14
15  // Nous devons ajouter au sillon les pixels voisins avec les plus petits coûts pour rejoindre
       le haut de l'image depuis le bas.
16  currentLine = cTable.height − 1
17  nvPixel = "Create a new pixel"
18
19  while currentLine ≥ 1
20       nvPixel = find_optimal_pixel(cTable, currentLine + 1,
                                      optimalGroove[currentLine + 1].column)
21       optimalGroove.path[currentLine].line = curentLine
22       optimalGroove.path[currentLine].column = nvPixel.column
23       currentLine = currentLine − 1
24
25  optimalGroove.cost = minLastLine
26  return optimalGroove
```

Les lignes 1 à 5 sont des initialisations de variables et de structures nécessaires pour plus tard. Les lignes 7 à 10 permettent de trouver le pixel avec le plus petit coût sur la dernière ligne de l'image. Les lignes 12 et 13 permettent d'ajouter le pixel avec le plus petit coût sur la dernière ligne au sillon optimal. A ce stade, nous avons un sillon démarrant au bas de l'image constitué de 1 pixel. Les lignes 19 à 23 permettent d'ajouter des pixels au sillon pour avoir un sillon qui va du haut vers le bas de l'image. Enfin, on initialise le coût associé au sillon et on retourne le sillon.

3.5 Pseudo-code pour renvoyer l'image réduite de k pixels en largeur

La fonction qui retourne l'image réduite de k pixels en largeur est la fonction REDUCEIMAGEWIDTH(image, k) spécifiée dans le fichier `slimming.h`. Le paramètre *image* correspond

à l'image qu'on souhaitre réduire et k le nombre de pixels qu'on souhaite retirer de la largeur de l'image.
La fonction COPY_PNM_IMAGE(source, dest) permet de copier l'image *source* dans l'image *dest*. On fait ceci dans le pseudo-code qui va suivre pour ne pas modifier l'image de départ.
La fonction COMPUTE_COST_TABLE(image) permet de calculer la table des coûts associés à chaque pixel de l'image passée en paramètre. L'élément (i, j) de la table correspond au coût associé au pixel (i, j) de l'image.
La fonction REMOVE_GROOVE_IMAGE(source, groove) permet de retirer le sillon *groove* de l'image *source*.
La fonction UPDATE_COST_TABLE(image, costTable, groove) permet de mettre à jour la table des coûts *costTable* sur base du sillon *groove* qui a été retiré de l'image sans la recalculer entièrement.

REDUCEIMAGEWIDTH($image, k$)

```
1  reducedImage = "Allocate a new PNMImage"
2  copy_pnm_image(image, reducedImage)
3     // On copie l'image d'origine dans une autre. Ensuite, on manipule la copie reducedImage.
4  nCostTable = compute_cost_table(reducedImage) // On calcule la table des coûts.
5
6  for number = 1 to k
7        optimalGroove = find_optimal_groove(nCostTable)
8        remove_groove_image(reducedImage, optimalGroove)
9        nCostTable = update_cost_table(reducedImage, nCostTable, optimalGroove)
10
11 return reducedImage
```

Les lignes 1 et 2 nous permettent d'obtenir une copie de l'image qu'on va manipuler pour ne pas modifier l'image source. C'est cette copie que la fonction va retourner. A la ligne 4, on calcule la table des coûts qui va être utile pour trouver les sillons qu'on doit supprimer de l'image. La table des coûts est la mémoïsation de l'approche par programmation dynamique. La ligne 7 permet de trouver le sillon optimal. Cette fonction a été expliquée en détail au point précédent. La ligne 8 nous permet de retirer de l'image les pixels se trouvant dans le sillon optimal. Ensuite, la ligne 9 met à jour la table des coûts sans la recalculer entièrement. On réalise aux lignes 7 à 9 un nombre k de fois ces opérations avec k qui correspond aux nombres de pixels qu'on souhaite retirer de la largeur de l'image.

3.6 Complexité en temps et en espace de la réduction en largeur d'une image

On a toujours n qui est la hauteur et m qui est la largeur de l'image. k correspond au nombre de pixels qu'on souhaite retirer de la largeur de l'image. Si on reprend le pseudo-code du point précédent en y ajoutant les complexités temporelles :

```
REDUCEIMAGEWIDTH(image, k)
 1  reducedImage = "Allocate a new PNMImage" // Θ(1)
 2  copy_pnm_image(image, reducedImage) // Θ(n × m)
 3  nCostTable = compute_cost_table(reducedImage) // Θ(n × m)
 4
 5  for number = 1 to k // k ×
 6        optimalGroove = find_optimal_groove(nCostTable) // Θ(n + m)
 7        remove_groove_image(reducedImage, optimalGroove) // O(n × m)
 8        nCostTable = update_cost_table(reducedImage, nCostTable, optimalGroove) // O(n × m)
 9
10  return reducedImage // Θ(1)
```

Si on additionne les complexités temporelles de chaque ligne, on obtient $\mathcal{O}((2+2k)(n \times m) + k \times (n+m) + 2)$ pour la complexité en temps de la fonction REDUCEIMAGEWIDTH.

Concernant les complexités spatiales :

- Stocker la copie de l'image qu'on va retourner représente une complexité spatiale de $\Theta(n \times m)$.
- Stocker la table des coûts représente une complexité spatiale de $\Theta(n \times m)$.
- Stocker le sillon optimal représente une complexité spatiale de $\Theta(n)$.

Si on additionne les complexités spatiales, on obtient une complexité $\Theta(2 \times (n \times m) + n)$ en espace.