# 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

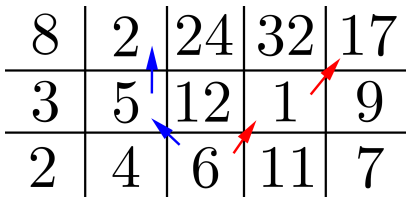


FIGURE 1 – Contre-exemple qui démontre la non-optimalité

# 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 $5 \times 3$, 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

# 5 Pseudo-code de l'implémentation efficace

```
1:  function DECODE(B, T)
2:      tab = "New array of double, of size m × n"
3:      moves = "New array of size_t, of size m × n"
4:      while currentBit ≤ B.nbBits do
5:          decoded = "Decode byte at currentBit with B, T"
6:          if decoded.character is endCharacter then
7:              break
8:          end if
9:          success = destination.add(decoded.character)
10:         if !success then
11:             return Erreur
12:         end if
13:     end while
14: end function
```

# 6 Pseudo-code de la réduction d'image

# 7 Complexité de la solution