AOD
Explorations d'images

Frédéric Wagner

2021

Dans ce sujet on s'intéresse à différents algorithmes sur des images bitmap ou chaque pixel est soit noir (vrai) soit blanc (faux).

Pour les deux parties du projet on se propose d'implémenter un même algorithme de trois manières différentes en changeant l'ordre de parcours des pixels utilisés.

Cette année on utilisera python (à titre expérimental (oui, vous êtes les cobayes)).

Le choix de python va nous permettre de nous concentrer sur les algos, de mieux contrôler les défauts de cache (qui sont logiciels et non matériels). On appréciera également l'utilisation des itérateurs afin de simplifier les développements.

Le point négatif est que l'on s'éloigne un peu de vraies applications pratiques car il est difficile de manier des données très grandes en python sans devenir également très lent.

On vous fournit une fonction *limitation_memoire* qui vous permet de simuler une machine avec une capacité mémoire très faible. Attention, elle limite également les processus fils. On se propose de *ralentir* notre code mais en échange de fonctionner même sous fortes contraintes de mémoire.

1 Images

On vous fournit deux codes différents pour la manipulation d'images. Dans tous les cas, on ne considèrera ici *que des images carrées*, de taille $n \times n$.

Le module *image* fournit une classe *Image* où les lignes sont codées par des bytearray. Ce codage permet d'avoir une consommation mémoire plus raisonnable que si chaque pixel était stocké dans son propre objet.

Sur cette classe on dispose d'une méthode de classe *aleatoire* qui permet la création d'une image où chaque pixel est tiré uniformément avec la probabilité donnée. Pour ce projet on se propose de travailler sur des images aléatoires avec une probabilité de 30% pour qu'un pixel soit noir.

On dispose également d'une méthode *pbm* permettant une sauvegarde de l'image dans un format lisible par un visualiseur et d'une méthode *affichage* réalisant un affichage ascii dans le terminal. On peut également sauvegarder ces images dans un fichier binaire à l'aide de la méthode *sauvegarde*.

Cette classe a une consommation mémoire quadratique et on cherche dans ce projet à limiter la consommation mémoire.

On vous fournit donc une seconde classe d'image dans le module *imagep*, la classe *ImageParesseuse*.

Pour cette classe, l'image est stockée sur le disque dans un format binaire (le même format que pour *Image*). La différence est qu'à un instant donné seule une partie du fichier est chargée en mémoire. Le fichier est décomposé en blocs et la méthode *bloc* de la classe *ImageParesseuse* utilise le décorateur *lru_cache* de *functools* que nous avons vu en première année. Le cache est ici paramétré pour une taille de 256 blocs.

L'implémentation est intéressante en elle-même, n'hésitez pas à jeter un coup d'œil.

2 Travail à réaliser

Il faut implémenter différents algorithmes et rendre sur teide le code ainsi qu'un mini-rapport à compléter.

On vous demande pour le rapport de générer un fichier *solution.html*. Tous les outils sont autorisés pour générer cette page mais le plus simple est de compléter le fichier *solution.md* fourni et de le compiler avec *pandoc --webtex solution.md -o solution.html*.

Pour une fois, on vous demande un html et non un pdf car la solution contient des fichiers gif animés.

2.1 Exploration

Premier algorithme : on part du pixel (0, 0) que l'on colorie en noir et l'on continue en propageant le coloriage sur tous les voisins blancs de proche en proche. On peut facilement programmer ça en récursif, mais nous vous proposons ici une version itérative, à l'aide d'une pile :

```
def exploration(image):
    """
    exploration (et modification)
    de l'image en profondeur d'abord,
    a l'aide d'une pile.
    on yield a chaque etape la pile.
    """
    def pixel_blanc(p):
        return not image.pixel(*p)
```

```
    a_voir = [(0, 0)]
    while a_voir:
        courant = a_voir.pop()
        if pixel_blanc(courant):
            image.noircir_pixel(*courant)
            yield a_voir
            a_voir.extend(filter(pixel_blanc, voisins(image.taille, courant)))
```

On vous fournit un fichier *exploration_aleatoire_pile.py* qui réalise l'exploration sur une image aléatoire et génère un fichier *gif* correspondant à l'exécution.

On vous demande d'utiliser le même algorithme mais en changeant l'ordre de parcours des pixels à l'aide d'un tas.

— une première version où on stocke simplement les pixels dans un tas au lieu d'une pile

— une seconde version où chaque pixel a une priorité obtenue en entrelaçant les bits de son abscisse et les bits de son ordonnée. Par exemple, le pixel (5, 22) a des bits 101 et 10110. En entrelaçant (poids faible d'abord) on obtient 1000111001 soit 569.

On vous demande dans le rapport :

— de dessiner une animation gif pour chaque algorithme (éviter les animations qui s'arrêtent dès le départ) ;

— d'expliquer quel est le meilleur ordre pour le parcours des voisins dans la fonction *utils.voisins* ;

— de tracer le max de la taille de la pile (ou du tas) pour des tailles croissantes d'images et donner votre avis ;

— de dessiner (à main levée) un contre-exemple atteignant une taille de tas quadratique.

2.2 Vue dégagée

Pour le second algorithme, c'est encore plus simple, le code est déjà écrit. Il s'agit d'un parcours des pixels qui cherche celui avec la meilleure vue. Chaque pixel blanc peut voir dans toutes les directions (haut, bas, gauche, droite) jusqu'à ce que son regard croise un pixel noir. On cherche le pixel qui voit le plus d'espace.

Le code est le suivant :

```
def vue_degagee(image, iterateur_pixels):
    """
    renvoie la position avec la meilleure vue
    """
    def pixel_blanc(pixel):
```

```
        return not image.pixel(*pixel)

    def vue_pixel(pixel):
        def vision(pixels):
            return takewhile(pixel_blanc, pixels)

        ligne, colonne = pixel
        haut = vision((l, colonne) for l in reversed(range(0, ligne)))
        bas = vision((l, colonne) for l in range(ligne+1, image.taille))
        vertical = chain(haut, bas)

        gauche = vision((ligne, c) for c in reversed(range(0, colonne)))
        droite = vision((ligne, c) for c in range(colonne+1, image.taille))
        horizontal = chain(gauche, droite)

        return compte(chain(horizontal, vertical))

    return max(filter(pixel_blanc, iterateur_pixels), key=vue_pixel)
```

On se propose cette fois-ci de travailler sur des images paresseuses. On peut par exemple limiter la mémoire avant le chargement de l'image.

Cet algorithme prend en entrée un itérateur sur les pixels et nous allons utiliser trois itérateurs différents.

— un itérateur ligne par ligne ;
— un itérateur par blocs ;
— un itérateur cache-oblivious à l'aide d'une z-curve. Pour celui-ci, vous pouvez utiliser un algorithme récursif (diviser pour régner) et *yield from*.

On vous demande de compléter les questions suivantes dans le rapport :

— en terme de défauts, les accès verticaux et horizontaux sont-ils équivalents ? (Justifiez)
— tracez une courbe des défauts de cache générés lors du calcul de la vue pour les trois itérateurs, sur des tailles d'images croissantes
— pouvez-vous expliquer les comportements des différents algorithmes ?
— expériences additionelles : si vous le voulez vous pouvez réaliser d'autres expériences qui vous semblent intéressantes.