TP C n°2

Remplissage par diffusion

Dans ce TP, nous allons codés l'algorithme *Flood fill* (remplissage par diffusion en français). Cet algorithme est notamment utilisé dans des logiciels d'édition photo comme Gimp ou Photoshop : il s'agit de l'outil « remplissage ».

Le but de ce TP est de réviser les parcours de graphes en impératif : le parcours en largeur et le parcours en profondeur. ***Dans ce TP, on n'utilisera* que *le style impératif, pas de récursif.***

I. Introduction.

On considère une image de taille 50×50 pixels. Cette image pourra être composée de trois couleurs : noir (représenté par l'entier `0`), gris (représenté par l'entier `1`) et violet (représenté par l'entier `2`).

On définit les types ci-dessous.

```c
typedef struct {
  int color;
  int i, j;
} pixel;

typedef pixel** image;
```

On pourra ajouter des champs à la structure `pixel` lors de la réalisation des parcours. Le champ `color` correspond à la couleur du pixel, et les entiers `i` et `j` représentent la position dans l'image.

L'algorithme *flood fill* consiste à réaliser un parcours de graphe, où le graphe est celui des pixels adjacents. On pourra passer d'un nœud au suivant *seulement si* les deux nœuds ont la même couleur. On représente l'exécution de l'algorithme sur la Figure 1.

Afin de réaliser les parcours, on s'aidera des structures de données dans les fichiers associés au TP.

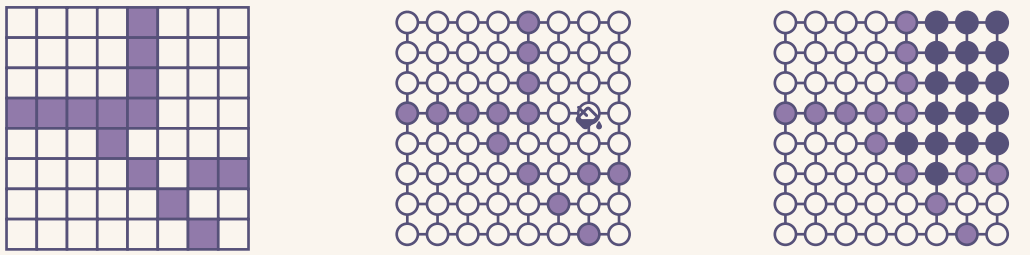

Image originelle — Graphe de l'image — Graphe après *flood fill*

Figure 1. Algorithme *flood fill*

On ne stockera pas le graphe de l'image *explicitement* mais, à partir d'un nœud, on pourra déterminer ses voisins. La représentation est dite *implicite*.

Q1. Coder une fonction `void show(image)` qui affiche dans la console une image donnée. On associera les couleurs à des caractères : blanc deviendra ` ` (un espace), noir deviendra `@`, et violet deviendra `+`. On ferra attention à l'orientation de l'image.

II. Représentation implicite du graphe.

Dans cette section, on construit la représentation implicite du graphe.

Q2. Coder une fonction `pixel* voisin_gauche(pixel*, image)` qui renvoie un pointeur sur le voisin gauche du pixel actuel. Si ce voisin n'existe pas (bordure de l'image), on renverra `NULL`.

Q3. Coder une fonction `pixel* voisin_droite(pixel*, image)` qui renvoie un pointeur sur le voisin droite du pixel actuel. Si ce voisin n'existe pas (bordure de l'image), on renverra `NULL`.

Q4. Coder une fonction `pixel* voisin_haut(pixel*, image)` qui renvoie un pointeur sur le voisin haut du pixel actuel. Si ce voisin n'existe pas (bordure de l'image), on renverra `NULL`.

Q5. Coder une fonction `pixel* voisin_bas(pixel*, image)` qui renvoie un pointeur sur le voisin bas du pixel actuel. Si ce voisin n'existe pas (bordure de l'image), on renverra `NULL`.

III. Parcours en largeur

Q6. Quelle structure de donnée utilise-t-on lors du parcours en largeur ?

En supposant votre réponse correcte, une implémentation en C de cette structure de données est disponible dans les fichiers associées au TP.

Q7. Implémenter une fonction `void floodfill_bfs(int i, int j, image img, int c)` qui exécute l'algorithme *flood fill* avec parcours en largeur en commençant du pixel `(i, j)` et qui remplira avec la couleur `c`.

> **Remarque.** On prendra soin de n'ajouter les nœuds qu'une seule fois dans la structure de donnée. Comme montré sur la Figure 2, lors du parcours depuis le nœud rouge, les nœuds en bleu peuvent être ajoutés plusieurs fois dans la structure de donnée si l'implémentation du parcours n'est pas correcte. On prendra donc soin de « peindre » le pixel *immédiatement après* l'avoir ajouté à la structure de donnée.

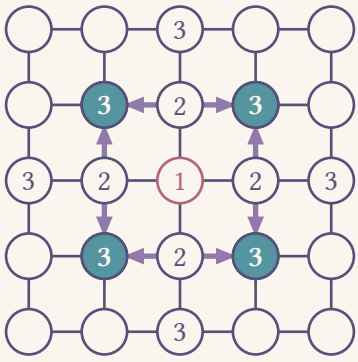

Figure 2. Attention à un ajout double

Q8. En utilisant la fonction codée en Q1, réaliser une animation de l'algorithme *flood fill* qui s'exécute sur une image test (Figure 3, page 3). On appellera la fonction `show` chaque fois qu'un pixel est « peint », puis on attendra un instant.

> **Indication.** Afin d'attendre 1 ms, on pourra appeler la fonction `wait1` définie dans le fichier `utils.c`. On pourra utiliser la fonction `clear_screen` présente dans le même fichier, afin d'effacer le texte dans la console.

IV. Parcours en profondeur

Q9. Quelle structure de donnée utilise-t-on lors du parcours en profondeur ?

En supposant votre réponse correcte, une implémentation en C de cette structure de données est disponible dans les fichiers associées au TP.

Q10. Implémenter une fonction `void floodfill_dfs(int i, int j, image img, int c)` qui exécute l'algorithme *flood fill* avec parcours en profondeur en commençant du pixel $(\texttt{i}, \texttt{j})$ et qui remplira avec la couleur `c`.

> **Remarque.** On prendra encore soin de « peindre » le pixel *immédiatement après* l'avoir ajouté à la structure de donnée.

Q11. En utilisant la fonction codée en Q1, réaliser une animation de l'algorithme *flood fill* qui s'exécute sur une image test (Figure 3).

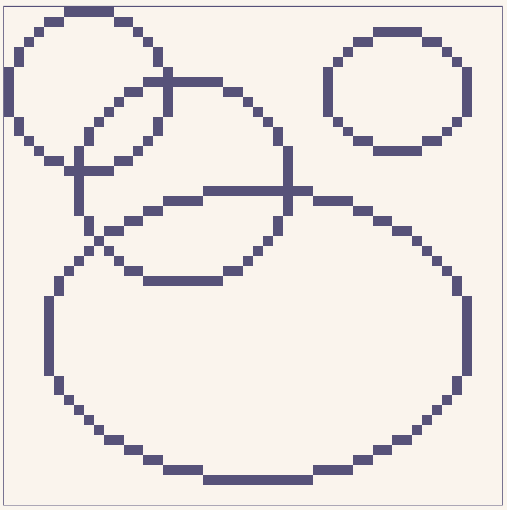

Figure 3. Image test

V. Fichiers associés au TP.

Dans le fichier `utils.c`, on retrouve :
- la fonction `wait` qui permet d'attendre un instant,
- la fonction `load_image` qui permet de charger une image d'un fichier comme `img1.txt`,
- la fonction `clear_screen` qui permet d'effacer la console.

Dans le fichier `stack.c`, on retrouve une implémentation d'une pile :
- la fonction `stack_create` qui permet de créer une pile,
- la fonction `stack_free` qui permet de libérer la pile (à n'appeler *que* sur une pile vide)
- la fonction `stack_is_empty` qui permet de tester si la pile est vide,
- la fonction `stack_push` qui permet d'ajouter un élément sur la pile,
- la fonction `stack_pop` qui permet de retirer le premier élément de la pile (et retourner cet élément),
- la fonction `stack_peek` qui permet d'accéder à l'élément en haut de la pile (sans modifier la pile).

Dans le fichier `queue.c`, on retrouve une implémentation d'une file :
- la fonction `queue_create` qui permet de créer une file,
- la fonction `queue_free` qui permet de libérer la file (à n'appeler *que* sur une file vide)
- la fonction `queue_is_empty` qui permet de tester si la file est vide,
- la fonction `queue_enqueue` qui permet d'ajouter un élément à la fin de la file,
- la fonction `queue_dequeue` qui permet de retirer le premier élément de la file (et retourner cet élément).

Dans le fichier `linked_list.c`, on retrouve des éléments de liste chaînées utilisés par les implémentations de la file et de la pile.

Le fichier `img1.txt` pest l'image test représenté sur la Figure 3.