Projet: "Le Robot Trieur" (Sorting Robot)

Nous considérons dans ce projet un jeu dit "combinatoire" appelé le Robot Trieur (Sorting Robot, en anglais). Vous pouvez trouver le descriptif de ce jeu à la page suivante. Une solution de ce jeu est évaluée en nombre de cases parcourues par le robot : l'objectif de ce projet est de déterminer des solutions où le robot se déplace le moins possible. Ainsi, il sera possible d'indiquer à un joueur de ce jeu s'il est proche ou loin des meilleurs scores possibles.

L'énoncé de ce projet se subdivise en deux parties :

- Partie 1 : Algorithme "au plus proche"

  Rendu de la partie 1 (code et solutions) lors du TME 8 (semaine 26/03/18)
- Partie 2 : Cas des grilles de jeu à une case par couleur

  Rendu de l'ensemble des deux parties lors du TME 11 (semaine 14/05/18)

Pour le rendu final, vous montrerez à votre chargé de TD votre code, son fonctionnement et ses performances (il y a un document sur le site du module précisant le contenu d'un rendu). Pour les étudiants qui ne peuvent être présents au TME11, vous devez contacter vos chargés de TD afin de prévoir une autre date au plus près de la semaine de rendu.

Chaque partie est divisée en exercices, qui vont vous permettre de concevoir progressivement le programme final. Chacun des exercices est le sujet des séances de TME de 3 à 11. Il est conseillé de suivre les étapes données par ces différents exercices, car chaque exercice est l'application directe de notions qui ont été introduites en cours et en TD en parallèle au projet.

Il est impératif de travailler régulièrement afin de ne pas prendre de retard et de pouvoir profiter des séances correspondantes pour chaque partie du projet.

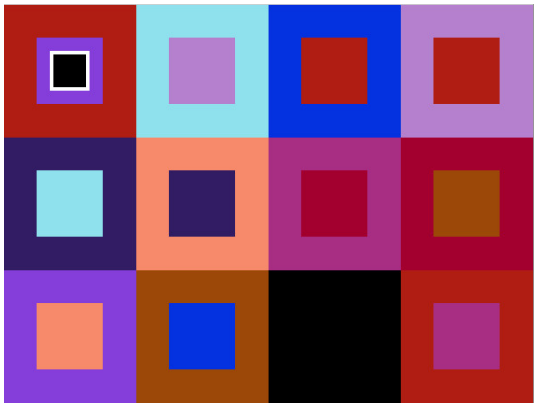
Remarque : Cet énoncé peut connaître des petites évolutions afin d'apporter des précisions ou des indications : venez en Cours/TD/TME pour les connaître et consulter fréquemment la page du module : `https://www.licence.info.upmc.fr/lmd/licence/2017/ue/2I006-2018fev/`

Cadre du projet

Le jeu dit du "Robot Trieur" (ou Sorting Robot, en anglais) se compose d'une grille de jeu et d'un robot pouvant se déplacer sur la grille de jeu. Chaque case possède une couleur de fond et et peut comporter une pièce colorée. Le but du jeu consiste à déplacer le robot de sorte que chaque pièce soit rangée sur une case de même couleur. A chaque fois qu'une pièce est amenée sur une case de même couleur, la case devient noire. Le but est donc d'obtenir un écran noir.

• *Grille de jeu*

Le jeu se déroule sur une grille de m lignes et n colonnes (potentiellement, m peut valoir 1 et la grille est alors une seule ligne de n colonnes). Dans tout le jeu, le noir correspond à une absence de couleur. Sur l'exemple ci-dessous, une grille de 3 lignes et 4 colonnes contient donc 12 cases. Chaque case peut contenir 0 ou 1 pièce. Une case possède une couleur de fond et peut posséder une pièce représentée par un rectangle au milieu de la case. Une pièce possède une couleur. Si une case ne comporte pas de pièce, elle comporte alors un rectangle noir.



Les lignes sont numérotées de 0 à $m-1$ et les colonnes de 0 à $n-1$. Dans l'exemple, la case $(1,0)$ est de fond bleu foncé et contient une pièce (rectangulaire) de couleur bleu-ciel. La pièce bleu ciel doit aller en case $(0,1)$ qui contient elle une pièce mauve qui doit aller en case $(0,4)$,... La case $(2,2)$ est noire car elle est de même couleur que la pièce qu'elle contient : il n'y a donc rien à faire pour cette case.

• *Actions du robot*

Le robot est au départ en case $(0,0)$: il est représenté par un petit rectangle coloré qui est entouré de blanc. Au départ, le robot ne porte pas de pièce et le robot est alors noir. S'il porte une pièce de couleur c , il devient un rectangle de couleur c entouré de blanc.

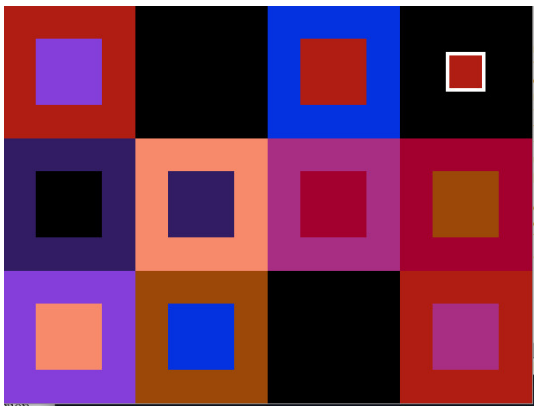
Un robot peut effectuer les *actions* suivantes :
- *Faire un pas* à gauche 'L', à droite 'R', en haut 'U' ou en bas 'D' : pour passer à une case à gauche, à droite, au-dessus ou au-dessous (sans sortir de la grille)
- *Faire un échange* (ou swap, en anglais) 'S' : cette action a plusieurs configurations :
    - Si le robot n'a pas de pièce et qu'il est sur une case avec une pièce : il prend alors la pièce.

- le robot a une pièce et qu'il est sur une case sans pièce : il dépose alors sa pièce ; le robot a une pièce et il est sur une case avec pièce : il échange sa pièce avec celle de la case.
- Si un robot est sans pièce sur une case sans pièce ou sur une case noire, il ne se passe rien.

Il est important de noter que le robot choisit librement d'effectuer l'une ou l'autre des actions : en particulier, rien ne l'oblige à déposer une pièce sur une case de même couleur (même si c'est à terme son objectif final).

Le but du jeu est de déplacer le robot et d'effectuer des échanges de manière à ce que toutes les pièces soient placées sur une case de même couleur. La case devient alors noire : à la fin du jeu, la grille entière doit donc être noire. L'objectif pour gagner est en plus de réaliser cela avec un nombre de pas le plus petit possible (le nombre d'échanges n'est pas pris en compte).

Une séquence d'actions du robot peut donc s'écrire comme une suite de lettres parmi L, R, U, D et S. Par exemple, la grille ci-dessous est obtenue à partir de la grille initiale plus haut par la séquence d'actions ('D', 'S', 'R', 'U', 'S', 'R', 'R','S'). A l'issue de cette séquence, la case $(1, 0)$ n'a plus de pièce, les cases $(0, 1)$ et $(0, 4)$ sont devenues noires. Le robot est à présent sur la case $(0, 4)$ et porte une pièce rouge.



• *Instances de grille*

On distingue deux types d'instances de grille :
- les instances où il y a plusieurs cases par même couleur.
- les instances où il y a une seule case par couleur.

Dans le cas de la grille ci-dessus, le robot a par exemple la possibilité de choisir d'aller déposer sa pièce rouge en case $(0, 0)$ ou en case $(2, 4)$ (notez qu'il peut aussi choisir de déposer sa pièce ailleurs temporairement).

• *Mais alors quand est-ce qu'on gagne ?*

Si le jeu se limitait à déterminer une séquence d'actions qui mènent à une grille toute noire, le jeu serait très facile. L'objectif est d'utiliser un nombre minimal de pas. Mais dans ce cas, comment peut-on savoir si on a mal ou bien joué ?

Le but de ce projet est justement de faire déterminer par un algorithme la séquence la plus courte possible : ainsi, un joueur pourra se "mesurer à l'ordinateur" pour avoir le meilleur score.

• *Est-ce un jeu difficile ?*

En fait ce jeu n'est pas qu'un jeu. Il est une simplification de déplacements possibles de robots en atelier ou en usine : cela peut-être des robots qui construisent ou réparent des cartes électroniques, ou des robots qui câblent des prises ethernet, ou même des robots géants qui déplacent des objets très lourds comme du combustibles nucléaires dans des centrales électriques.
Ce jeu a été étudié par des chercheurs. Il a été prouvé qu'il est "difficile" de trouver la séquence la plus courte possible : il s'agit d'un problème NP-difficile. Quand un problème est NP-difficile, cela veut dire qu'on ne connaît pas d'algorithme polynomial pour le résoudre exactement (si vous en trouvez un pour ce jeu, vous serez célèbre !). Cela n'empêche pas qu'il est possible de trouver de très bonnes solutions en temps "raisonnable" !

Description du code du jeu

Vous pouvez téléchargez sur le site de l'UE une archive contenant les fichiers suivants :
- `Grille.h` et `Grille.c`
- `Solution.h` et `Solution.c`
- `API_AffGrille.h` et `API_AffGrille.c`
- `entree_sortie.h` et `entree_sortie.c`
- `Game_SortingRobot.c`
- `Checker_SortingRobot.c`
- `Makefile`

Le Makefile permet de compiler l'ensemble du code.
Il y a deux exécutables `Game_SortingRobot` et `Checker_SortingRobot`.

• `Game_SortingRobot`

Le programme `Game_SortingRobot` prend en paramètre :
- le nombre m de lignes
- le nombre n de colonnes
- le nombre *nbcoul* de couleurs
- et une *graine* de génération aléatoire

Vous pouvez ainsi jouer au "Robot Trieur" !

Pour jouer, il vaut mieux étendre au maximum la fenêtre du terminal. En effet, le terminal vous indiquera alors la séquence que vous avez jouée et le score (nombre de pas du robot) effectué au fur et à mesure. Une fenêtre graphique s'ouvre devant le terminal. La gestion des touches est tout simplement :
- les flèches pour déplacer le robot
- la touche espace pour le "Swap"
- Escape ou clique sur l'icône de fermeture pour sortir du jeu avant la fin.

L'affichage de la grille est permis par l'API `API_AffGrille` qui utilise la SDL (Simple DirectMedia Layer) qui est une bibliothèque libre de gestion graphique et événementielle en langage C. N'hésitez pas à regarder le code correspondant qui est très lisible.

• `Les instances`

Les instances sont générées aléatoirement... En informatique, le "hasard' est généré par un algorithme déterministe qui "ressemble" au hasard. Ainsi si vous fournissez au programme des paramètres $(n, m, nbcoul, graine)$ donnés, vous obtiendrez toujours la même grille. Par exemple, la grille de l'exemple des pages précédentes est obtenue en utilisant $(3, 4, 10, 0)$.
On distingue deux types d'instances de grille :
- les instances où il y a plusieurs cases par même couleur : il suffit d'utiliser le programme avec $ncoul < m * n$,
- les instances où il y a une seule case par couleur : il suffit d'utiliser le programme avec $nbcoul = m*n$.

• `Grille.h` et `Grille.c`

Voici le fichier `Grille.h`

```
#ifndef __GRILLE__
#define __GRILLE__
#include "Solution.h"


// Type struct d une case d une grille
typedef struct {

  int fond;   // Entier couleur de 0 a nbcoul-1

  int piece;  // Entier couleur de 0 a nbcoul-1 si piece presente
              // -1 si pas de piece

  int robot;  // -2 si pas de robot sur cette case
              // -1 si robot present mais sans piece
              // Entier couleur de 0 a nbcoul-1 si robot present avec piece
} CCase;


// Type struct d une grille de jeu
typedef struct {
  // Un couple (i,j) donne des coordonnees ieme ligne, jeme colonne

  int m;          // Nb de lignes

  int n;          // Nb de colonnes

  int nbcoul;     // Nb de couleurs

  CCase ** T;     // Tableau a deux dimensions de la grille

  int ir, jr;     // Coordonnee du robot sur une case (dite case courante)

  int cptr_noire; // Compteur de cases noires

} Grille;
```

```
37
38
39 // Fonction realisant l'allocation d'une grille
40 // Les champs m,n  doivent deja etre remplis auparavant
41 void Grille_allocation(Grille *G);
42
43 // Fonction de generation d'une instance en fonction d une graine
44 // de generation aleatoire
45 void Gene_Grille(Grille *G, int graine);
46
47 // Fonction de changement de robot de la case courante
48 // a une case i,j
49 void changement_case(Grille *G, int i, int j);
50
51 // Fonction de prise, depot ou echange d une piece par le robot
52 // sur la case courante
53 void swap_case(Grille *G);
54
55 #endif
```

Le type struct `Grille` et le type `CCase` des cases de la grille contiennent ainsi toute l'instance. Vous pouvez trouver dans les commentaires, les choix faits pour cet encodage.

• `Solution.h` et `Solution.c`

Le struct `Solution` propose une liste de `char` simplement chaînée avec insertion en fin en $O(1)$. Cette liste permet de stocker des séquences de caractères 'L', 'R', 'U', 'D', 'S' correspondant aux actions du robot. On dit qu'une séquence est *valide* si elle ne conduit pas le robot en dehors de la grille et qu'elle est une *solution* si elle colorie la grille en noire en plaçant toutes les pièces sur une case de même couleur. Ce struct contient un compteur de pas qui décompte (uniquement) le nombre d'actions de déplacement.
`Solution.c` propose différentes fonctions de manipulation dont des Entrées/Sorties sur disque selon un format très lisible. Noter que ce format contient les données (m,n,$nbcoul$,$graine$) permettant de générer la grille correspondante.

• `Checker_SortingRobot`

Le programme `Checker_SortingRobot` prend en paramètre un fichier solution issue des méthodes de `Solution.h` et vérifie si la séquence contenue est bien valide, est bien solution et décompte son nombre de pas.
On peut l'utiliser en version rapide sans affichage ou alors avec une animation graphique : pour cela, le paramètre suivant est 0/1 pour provoquer l'affichage et le troisième paramètre est un temporisateur de l'animation.

PARTIE 1 : Algorithme "au plus proche"

Dans cette première partie, on considère des instances générales de grilles où une même couleur peut apparaître dans plusieurs cases. On s'intéressera même plutôt aux instances où le nombre de couleurs est petit devant le nombre de cases de la grille.

Dans ces grilles où une couleur apparaît plusieurs fois, le robot peut hésiter entre plusieurs cases de couleur c pour y transporter une pièce de couleur c. On se propose dans cette partie d'implémenter plusieurs versions du principe suivant, dit "au plus proche" :

Algorithme "au plus proche"
Réitérer tant qu'il reste une case non noire
- *Si le robot ne porte pas de pièce, il va prendre la pièce de la case la plus proche,*
- *Sinon le robot va placer sa pièce sur la case de même couleur que sa pièce et qui est la plus proche (en nombre de cases à parcourir).*

S'il existe plusieurs cases à même distance dans la grille, le robot choisit toujours la case "en haut, à gauche", c'est-à-dire la case de plus petit indice en ligne, et s'il y a plusieurs sur cette ligne celle de plus petit indice en colonne.

Les exercices suivant vont conduire à plusieurs implémentations de cet algorithme.

Q 0.1 Le premier exercice de ce projet consiste à télécharger l'archive du jeu, de le compiler, de jouer et de bien comprendre le début de cet énoncé.

Exercice 1 – TME4 (semaine 19/02/18) - Version naïve

Dans ce premier exercice, nous allons implémenter une version simple de l'algorithme "au plus proche" : on appellera cette version "naïve". Elle consiste pour déterminer quelles sont les cases candidates pour déplacer le robot, à parcourir la grille depuis la case $(0, 0)$ en parcourant toutes les cases à la recherche de la plus proche de la case courante.

On appelle chemin de la case (i, j) à une case (k, l) une séquence d'action de déplacement permettant de déplacer le robot situé en case (i, j) jusqu'à la case (k, l).

Q 1.1 (Bonus) Prouver que, d'une case (i, j) à une case (k, l),
- le chemin se déplaçant de $|k - i|$ cases verticalement vers (k, l), puis de $|l - j|$ cases horizontalement vers (k, l),
- le chemin se déplaçant de $|l - j|$ cases horizontalement vers (k, l), puis de $|k - i|$ cases verticalement vers (k, l),

sont des plus courts chemins.
Indication : Cette preuve peut-être faite par récurrence sur le nombre de lignes $|k - i|$ séparant les deux cases.

Q 1.2 Implémenter la fonction `void PlusCourtChemin(Solution *S, int i, int j, int k, int l)` qui prend en paramètre une `Solution` S et qui lui ajoute une séquence de caractère correspondant à un plus court chemin dans la grille allant de la case (i, j) à la case (k, l).

Q 1.3 Indiquer comment savoir, à partir d'une variable G de type `Grille`,
a) comment savoir si une case est noire (c'est-à-dire si elle porte une pièce de même couleur)
b) comment savoir si une pièce n'est pas noire
c) comment savoir si le robot porte une pièce
d) comment savoir, dans le cas où le robot porte une pièce, quelle couleur de pièce porte-t-il

Q 1.4 Implémenter une fonction `RechercheCaseNaif_c(Grille *G, int c, int i, int j, int *k, int *l)` qui retourne la case (k, l) de couleur c qui soit la plus proche de (i, j) au sens de l'algorithme "au plus proche". Même question avec la fonction `RechercheCaseNaif_nn(Grille *G, int i, int j, int *k, int *l)` qui recherche une case non-noire et possédant une pièce.

Q 1.5 Implémenter une fonction `algorithme_naif(Grille *G, Solution *S)` qui implémente la version naïve de l'algorithme "au plus proche". Cette fonction doit écrire sur disque la solution obtenue.

Q 1.6 Tester votre fonction sur des grilles de petites tailles pour tester sa validité expérimentale. Tester sur des grilles de plus en plus grandes pour déterminer jusqu'à quel ordre de grandeur (en nombre de cases) la fonction s'exécute en moins de 30 secondes. Tester auprès de vos encadrants si votre code trouve bien la solution attendue : il vous servira de référence pour valider expérimentalement les améliorations à venir dans le projet.

Q 1.7 Avez-vous bien organisé votre code ? Etant donné qu'il y a plusieurs méthodes dans cette partie 1, une bonne façon d'organiser votre code est de créer un programme main qui prend en ligne de commande les paramètres d'une instance, ainsi que la méthode que vous allez lancer. Par exemple, la version naïve a le numéro de méthode 1. Organiser correctement votre code avant de passer à l'exercice suivant.

Exercice 2 – TME 5 (semaine 26/02/18) - Version circulaire

Une autre version moins naïve est de rechercher, circulairement autour du robot, la case la plus proche où il doit se rendre. Attention, on veut respecter le fait qu'entre plusieurs cases proches, on choisit la case la plus en haut à gauche.

On peut remarquer qu'étant donnée une distance de $L \geq 1$ cases, les cases à distance L de la case (i, j) sont données par le principe suivant :

$k \leftarrow i - L$
$lg \leftarrow j$
$ld \leftarrow j$
`Tant que` $k \leq i$ `Faire`
    `Afficher` (k, lg) `et` (k, ld)
    $i \leftarrow i + 1$
    $lg \leftarrow lg - 1$
    $ld \leftarrow ld + 1$
`FinTanque`
$k \leftarrow i + 1$
$lg \leftarrow j - L + 1$
$ld \leftarrow j + L - 1$

`Tant que` $k \leq i + L$ `Faire`
    `Afficher` (k, lg) `et` (k, ld)
    $i \leftarrow i + 1$
    $lg \leftarrow lg + 1$
    $ld \leftarrow ld - 1$
`FinTanque`

Pour cette valeur de L, les cases sont en plus rangées dans l'ordre donnant en les cases les plus en haut à gauche d'abord. Bien sûr, il faut tester à chaque si ces cases existent dans la grille et correspondent à la case recherchée.

Q 2.1 En utilisant le principe "recherche circulaire", donner une nouvelle version des fonctions `RechercheCaseCirculaire_c(Grille *G, int c, int i, int j, int *k, int *l)` et `RechercheCaseCirculaire_nn(Grille *G, int i, int j, int *k, int *l)` comme dans la méthode naïve. En déduire un méthode circulaire `algorithme_circulaire(Grille *G, Solution *S)`.

Q 2.2 Tester votre fonction en vérifiant si elle fournit exactement le même résultat que la méthode naïve (qui a été validé à l'exercice précédent). Jusqu'à quel nombre de cases cette méthode fonctionne-t-elle en moins de 30 secondes ? Comparez les deux méthodes en mesurant le temps CPU par la commande **clock**.

Q 2.3 Pour évaluer la complexité des méthodes, nous allons nous placer dans le cas des grilles carrées avec $m = n$. Quelle est la complexité (pire-cas) de ces deux méthodes en fonction du nombre n de lignes et de colonnes ? Est-ce que vos expérimentations permettent de retrouver cette évaluation de la complexité ?

Exercice 3 – TME 5-6 (semaines 26/02/18 et 05/03/18) - Version "par couleur"

L'idée de cette méthode, dite "par couleur" est de remarquer que, lors de la recherche d'une case de couleur c, il est possible de limiter la recherche aux cases de couleurs c. Afin de limiter cette recherche, il est possible d'effectuer un prétraitement avant l'algorithme proprement dit qui construit une structure de données où les cases sont rangées par couleurs.

Q 3.1 Construire une bibliothèque pour une liste doublement chaînée `LDC` (voir TD2) où chaque cellule `CelluleLDC` de la liste contient des coordonnées (i, j) d'une case de grille :

```
1  typedef struct celluleLDC{
2    int i,j;
3    struct celluleLDC* prec; /* pointeur sur l'element precedent de la liste */
4    struct celluleLDC* suiv; /* pointeur sur l'element suivant de la liste */
5  } CelluleLDC;
6
7  typedef struct {
8    CelluleLDC* premier;  /* Pointeur sur element en tete */
9    CelluleLDC* dernier;  /* Pointeur sur element en fin */
10 } LDC;
11
12 CelluleLDC* creerCellule(int i, int j); // qui alloue et retourne une cellule
13 void LDCInitialise(LDC *ldc);           // qui initialise une liste
14 int LDCVide(LDC* ldc);                  // qui teste si la liste est vide
15 void LDCInsererEnFin(LDC* ldc, int i,int j); //qui insere une nouvelle cellule en fin
16 void LDCenleverCellule(LDC* ldc, CelluleLDC * cel);
17      // qui supprime une cellule a partir d un pointeur sur la cellule
```

```
18 void LDCafficher(LDC* ldc);   // un affichage en cas de besoin pour debugage
19 void LDCdesalloue(LDC *ldc);
20      // qui desalloue toute la liste (si elle n est pas vide a la fin)
21 }
```

Q 3.2 Ajouter une fonction `CelluleLDC* LDCrechercherPlusProcheCase(LDC* ldc, int i, int j);` qui recherche dans une liste la case la première case la plus proche de (i, j).

Q 3.3 On veut utiliser un tableau `LDC *TC` de listes `LDC` : chaque case $TC[c]$ pour c allant de 0 à $nbcoul-1$ correspond à une liste doublement chaînée des cases de couleurs c. Donner une fonction `void algorithme_parcouleur(Grille *G, Solution *S);` qui implémente la méthode "au plus proche". Cette fonction commence par allouer et intialiser le tableau `TC` et le remplit en parcourant la grille de haut en bas et de droite à gauche : ainsi, la fonction `LDCrechercherPlusProcheCase` retournera alors la case la plus proche la plus en haut à gauche. Pour cette fonction, utilisez bien sûr la fonction de la question précédente pour rechercher dans `TC` la case la plus proche d'une couleur donnée mais vous pouvez continuer à utiliser une fonction des exercices précédents pour trouver une case non-noire et possédant une pièce.

Q 3.4 Tester et valider votre fonction à partir des fonctions validées des exercices précédents (vous devez retrouver exactement la même séquence). Jusqu'à quelle taille en nombre de cases, cette méthode permet de résoudre des grilles en moins de 30 secondes. Comparez les 3 méthodes.

Q 3.5 En se plaçant à nouveau dans le cadre d'une grille carrée avec $n = m$, donner la complexité (pire-cas) de cette méthode en fonction de n. Même question en utilisant le "paramètre caché" de l'instance suivan : α le nombre maximal de pièces d'une même couleur.

Exercice 4 – TME 6-7 (semaines 05/03/18 et 19/03/18) - Version "par AVL"

Remarque importante : le stockage mémoire de la liste chaînée Solution devient vite trop important pour la mémoire des ordinateurs, vous pouvez la déconnecter pour la suite de la Partie 1 pour comparer les méthodes "Par Couleur" et par "AVL" sur des grandes tailles d'instance.

La méthode dite "Par AVL" consiste à remarquer que rechercher une case la plus proche d'une couleur c sur une ligne i donnée peut se faire par un AVL.

[Des précisions seront données prochainement sur cet exercice].

Q 4.1 Construire une structure d'arbre binaire de recherche équilibré selon la technique AVL pour stocker des entiers.

Q 4.2 Construire une structure de tableau M à double indice telle que `M[c][i]` correspond à un AVL donnant toutes les indices j tels que la case (i, j) contient une case de couleur c.

Q 4.3 Ajouter à la structure d'AVL une méthode en $O(log(\beta)$ indiquant dans l'AVL `M[c][i]` la case la plus proche d'une case (k, l) donnée, où β est le nombre de cases dans l'AVL.

Q 4.4 Donner une fonction implémentant la méthode au plus proche avec cette structure.

Q 4.5 Tester et comparer cette version avec la version "Par Couleur".

Q 4.6 Donner la complexité de cette version.

RENDU TME 8 (semaine du 26/03/18) : Vous devez rendre le travail réalisé pour les parties 1 et 2 au TME 8. Pour cela, vous enverrez à votre chargé de TD un fichier .tar ou .zip contenant les fichiers sources, les fichiers réseaux que vous avez réussi à reconstruire à partir des chaînes, ainsi que leurs images générées avec Gnuplot. Le fichier contiendra également les courbes que vous avez obtenues lors de l'exercice 5. Votre code doit pouvoir être testé facilement (menu en console).
Nous vous incitons également à commencer le rapport final qui sera à rendre en fin de projet au TME 11.