

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.

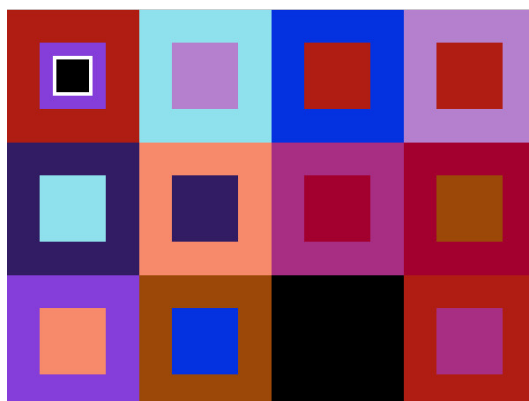
Remarque : Cet énoncé peut connaître des petites évolutions afin d’apporter des précisions ou des indications : venez en Cours/TD/TME pour les connaître et consulter fréquemment la page du module : <https://www.licence.info.upmc.fr/lmd/licence/2017/ue/2I006-2018fev/>

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 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 :

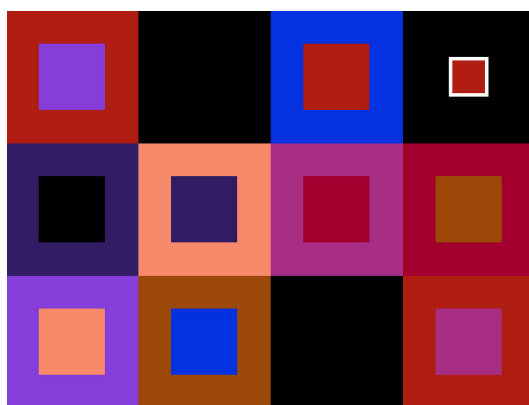
- *Faire un pas* à gauche ‘L’, à droite ‘R’, en haut ‘U’ ou en bas ‘D’ : pour passer à une case à gauche, à droite, au-dessus ou au-dessous (sans sortir de la grille)
- *Faire un échange* (ou swap, en anglais) ‘S’ : cette action a plusieurs configurations :
 - Si le robot n’a pas de pièce et qu’il est 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 télécharger 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 $(n, m, nb\text{coul}, graine)$ 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 $nb\text{coul} < m * n$,
- les instances où il y a une seule case par couleur : il suffit d’utiliser le programme avec $nb\text{coul} = m * n$.

• Grille.h et Grille.c

Voici le fichier `Grille.h`

```

1  #ifndef __GRILLE__
2  #define __GRILLE__
3  #include "Solution.h"
4
5
6  // Type struct d une case d une grille
7  typedef struct {
8
9      int fond;    // Entier couleur de 0 a nbcout-1
10
11     int piece;   // Entier couleur de 0 a nbcout-1 si piece presente
12                // -1 si pas de piece
13
14     int robot;   // -2 si pas de robot sur cette case
15                // -1 si robot present mais sans piece
16                // Entier couleur de 0 a nbcout-1 si robot present avec piece
17 } CCase;
18
19
20 // Type struct d une grille de jeu
21 typedef struct {
22     // Un couple (i,j) donne des coordonnees ieme ligne, jeme colonne
23
24     int m;        // Nb de lignes
25
26     int n;        // Nb de colonnes
27
28     int nbcout;   // Nb de couleurs
29
30     CCase ** T;   // Tableau a deux dimensions de la grille
31
32     int ir, jr;   // Coordonnee du robot sur une case (dite case courante)
33
34     int cptr_noire; // Compteur de cases noires
35
36 } 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)

$k \leftarrow k + 1$

$lg \leftarrow lg - 1$

$ld \leftarrow ld + 1$

FinTantque

$k \leftarrow i + 1$

$lg \leftarrow j - L + 1$

$ld \leftarrow j + L - 1$


```

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

```

Pour cette valeur de L , les cases sont en plus rangées dans l'ordre donnant 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 LDCInsérerEnFin(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 $nb\text{coul}-1$ 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 initialiser 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 suivant : α le nombre maximal de pièces d’une même couleur.

Exercice 4 – TME 6-7-8 (semaines 19/03/18 et 26/03/18) - 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.

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

Q 4.2 Valider votre structure d’arbre binaire de recherche équilibré : pour cela créer un programme Test qui ajoute et supprime de nombreux éléments. Une façon simple de concevoir ce programme est de générer aléatoirement des nombres entiers et de décider aléatoirement soit de l’ajouter, soit de le rechercher (et de le supprimer s’il existe).

Q 4.3 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 4.4 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 de l’AVL.

Q 4.5 Donner une fonction implémentant la méthode “au plus proche” avec cette structure.

Q 4.6 Tester et comparer cette version avec la version “Par Couleur”.

Q 4.7 Donner la complexité de cette version.

RENDU TME 8 (semaine allant du vendredi 02/04/18 au mercredi 11/04/18) : 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.

PARTIE 2 : Grilles de jeu à une case par couleur

Dans cette seconde partie, nous allons considérer plus particulièrement :

- le cas de grilles où une couleur apparaît une seule fois en tant que couleur de fond d'une case et une seule fois en tant que couleur de pièce : on appellera ce cas "une case par couleur" ;
- le cas de grille limitée à un simple vecteur, c'est-à-dire pour lequel $m = 1$.

En effet, dans le cas cumulé d'un vecteur avec "une case par couleur", rechercher le parcours du robot possédant le moins de pas possible a été prouvé par des chercheurs comme étant un problème facile : c'est-à-dire que l'on peut déterminer un tel parcours en temps polynomial.

Toutefois, nous allons implémenter cette méthode en essayant le plus possible d'avoir des structures et des fonctions qui ne sont pas spécifiques au cas d'un vecteur avec "une case par couleur" : ainsi nous obtiendrons des outils génériques qui nous permettront également de résoudre le cas général d'une grille avec un nombre de couleurs quelconque (mais pas nécessairement en déterminant le plus petit parcours).

Exercice 5 – TME 9 (semaines 04/04/18 et 09/04/18) - Graphe et Circuit

Q 5.1 Récupérer la nouvelle archive correspondant à la Partie 2.

Cette archive contient plusieurs nouveaux fichiers correspondant aux structures suivantes :

- **ListeDC.h** et **ListeDC.c** proposent un struct **LDC** d'une liste doublement chaînée de paires (i, j) d'entiers avec des méthodes d'insertion en tête, en fin, affichage et désallocation
- **Graphe.h** et **Graphe.c** une structure de graphe **Graphe** permettant de manipuler autrement une instance de grille.

Graphe des déplacements Pour une instance de grille G donnée, on considère un graphe orienté $H = (V, A)$ défini de la façon suivante :

- l'ensemble des sommets V correspondant aux cases (i, j) de la grille G ,
- l'ensemble des arcs A correspond à des arcs allant du sommet-case (i, j) au sommet-case (i', j') tels que la case (i, j) est non-noire et qu'elle contient une pièce ayant la même couleur que la couleur du fond de case (i', j') .

En d'autres termes, le graphe H indique les trajets possibles du robot permettant de placer une pièce sur une bonne case. Il faut noter qu'un arc représente un ensemble de déplacements sur la grille : en effet le robot doit traverser toutes les cases permettant d'instancier le trajet représenté par un arc.

Structure de graphe La structure de données implémentant le graphe H est une structure dite par "Liste d'adjacence". Elle se compose :

- d'un struct **Graphe** qui contient une matrice de taille $m \times n$ de pointeurs sur un struct **Sommet** : ainsi on peut en $O(1)$ accéder au sommets représentant une case (i, j) .
- d'un struct **Sommet** qui contient les coordonnées (i, j) d'une case de la grille et une liste chaînée de struct **Arc**.
- d'un struct **Arc** qui est un élément d'une liste chaînée. Chaque élément donne un pointeur sur **Sommet** correspondant à l'autre extrémité de l'arc.

La classe `Graphe` possède plusieurs méthodes dont principalement :

- une méthode `Graphe_creation` de création du graphe H à partir d'une grille G : cette méthode alloue la structure de graphe en créant les sommets et les arcs respectant la définition.
- une méthode d'affichage du graphe.

Q 5.2 Créer un main comme dans la partie 1 qui prend en ligne de commande les données créant une grille de jeu ; et qui crée et affiche le graphe. Tester-le pour des petites grilles (en effet, l'affichage est rapidement peu lisible).

Permutation et circuit de permutation Une remarque importante est celle de la définition d'une *permutation* sur une grille de jeu. On appelle permutation des pièces le fait d'avoir permuté (c'est-à-dire échangé deux à deux de manière itérative) des pièces entre elles sur une grille. On peut noter alors qu'une grille de jeu peut être obtenue à partir d'une grille initiale où chaque pièce était placée sur une case de sa propre couleur : les pièces déplacées par la permutation seront alors soit sur une case d'une autre couleur, soit sur une case de sa propre couleur (et est alors noire).

Il est bien connu (voir ouvrage traitant des permutations) que l'action permettant d'obtenir cette permutation peut être décomposée en "circuits de permutation" : une pièce est emmenée à la place d'une autre, elle-même déplacée à la place d'une autre,... jusqu'à placer une pièce sur la case de départ.

Circuit de H En fait, les circuits de permutation sont exactement des circuits du graphe H . On choisit d'encoder un circuit de H comme une suite de paires (i, j) correspondant à des sommets-cases dans l'ordre d'un circuit. Pour un graphe H donné, on peut ainsi déterminer un ensemble de circuits tels que chaque sommet-case non-noire apparaît dans exactement un circuit : on dit que cet ensemble de circuit *couvre* les sommets non-noirs.

L'algorithme est assez simple (plus simple qu'un algorithme de parcours) :

- on utilise le champs `visit` du struct `Graphe` pour indiquer si un sommet est : -2 noire, -1 non visité, 0 si visité.
- on initialise les sommets à -2 si noire et -1 sinon.
- on explore la matrice case à case en cherchant s'il existe un sommet non-visité (à -1) : si c'est le cas, on lance une fonction de recherche d'un circuit à partir de ce sommet. Cette fonction marque à 0 tous les sommets du circuit.

Recherche d'un circuit La fonction recherche d'un circuit à partir d'un sommet i consiste à choisir arbitrairement un sommet j successeur non-visité de i (et non noire), puis de réitérer la même itération à partir de j tant que j est différent sur sommet initial (fonction récursive ou itérative).

Q 5.3 Donner une fonction `Graphe_Rech_Circuit(Graphe *H)` qui permet d'afficher un ensemble de circuits couvrant les sommets non-noirs de H .

Encodage des circuits de H Un circuit peut être encodé par une liste LDC. Comme on va devoir stocker plusieurs circuits de manière à couvrir tous les sommets non-noirs, on peut utiliser une structure de liste chaînée (nommée par exemple `Lcircuit`) dont chaque cellule contient une liste LDC. Cette liste doit être codée de manière à permettre une insertion en fin en $O(1)$. On peut par exemple suivre le code C suivant où `dernier` pointe sur l'élément de queue de la liste. (Les champs `jmin` et `jmax` seront utilisés dans l'exercice suivant).

```

1 typedef struct cell_circuit{
2     LDC L;
3     int jmin, jmax;
4     struct cell_circuit *suiv;
5 } Cell_circuit;
6
7 typedef struct{
8     int nb_circuit;
9     Cell_circuit *premier;
10    Cell_circuit *dernier;
11 } Lcircuit;

```

Q 5.4 Transformer la fonction `Graphe_Rech.Circuit(Graphe *H, Lcircuit *LC)` pour qu'elle retourne une liste de circuits de H couvrant les sommets-case non-noirs.

Exercice 6 – TME 9-10 (sem. 30/04/18 et 07/05/18) - Vecteur avec “une case par couleur”

Pour le cas très particulier des vecteurs (grille avec $m = 1$) avec “une case par couleur”, nous allons utiliser un algorithme polynomial proposé par Daniel Graf¹ qui fournit une solution ayant un nombre minimal de pas. Notez qu'il existe une extension de cet algorithme au cas où les cases sont reliées comme dans un arbre (donc ne sont pas une grille). En revanche, le cas général d'un assemblage planaire de cases est un problème NP-difficile. Pour avoir une idée de la preuve de validité et d'optimalité de cet algorithme dans le cas d'un vecteur, voir l'annexe de ce document.

Pour cet algorithme, nous avons besoin à l'issue de l'exercice précédent :

- i) des circuits de permutation de H dont le premier sommet correspond à la case la plus à gauche du vecteur : on note $jmin$ cet indice ;
- ii) d'une liste LC des circuits rangés par $jmin$ croissant ;
- iii) pour chaque circuit, on dispose de l'indice $jmax$ de la case la plus à droite de ce circuit.

Q 6.1 Construisez une fonction `void CalculJminJmax(Lcircuit *LC)` à l'issue de laquelle LC est triée et contient les valeurs $jmin$ et $jmax$.

Noter qu'à l'issue de l'exercice 1, vous avez sans doute déjà les items i) et ii) sans travail supplémentaire : il suffit, lors de la recherche des circuits d'explorer les cases du vecteurs de gauche à droite.

L'algorithme de Daniel Graf est le suivant :

INPUT : une grille vecteur avec une case par couleur

- Créer le graphe H
- Créer la liste LC des circuits de H
[Notez qu'elle est unique dans le cas “une case par couleur”]
- Calculer $jmin$ et $jmax$ pour chaque circuit
- Faire en sorte que chaque circuit commence par sa case $jmin$
- Trier la liste LC des circuits selon les $jmin$ croissants
- Créer une solution S vide

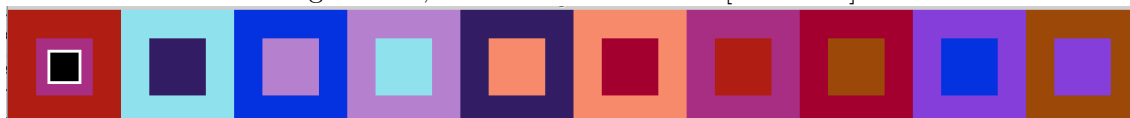
1. *How to Sort by Walking on a Tree*. Daniel Graf. Algorithms ESA, Lecture Notes in Computer Science LNCS 9294, p643-655 (2015)

- Créer un tableau de pointeurs *Tref* de n cases NULL.
[Au cours de cet algorithme Tref[j] contient NULL si la case i du vecteur n'a pas été rencontrée jusqu'ici, ou il contient un pointeur de l'élément de la solution S qui amène pour la dernière fois le robot à la case j lors du parcours correspondant à S]
 - Initialiser un indice de case *Jdroite* à 0.
[Au cours de cet algorithme, Jdroite contiendra l'indice de la case la plus à droite visitée par le robot.]
 - Pour chaque circuit *C* de *LC* dans l'ordre des *jmin* croissants
 - Si *Tref[jmin]* est NULL Alors
 - On met *Drapeau* à vrai et on sauvegarde *Jdroite* dans *JdroiteSav*
 - Insérez une séquence de déplacements "R" allant de *Jdroite* à *C->jmin* dans la solution *S* à la suite du pointeur *Tref[Jdroite]*
 - Ne pas oublier de mettre à jour *Jdroite* et *Tref*
 - [Drapeau sert à se rappeler qu'il faut revenir à JdroiteSav après ce déplacement à droite]*
 - FinSi
 - Ajouter une séquence de déplacements "S", "L" et "R" correspondant au circuit *C* dans la solution *S* à la suite du pointeur *Tref[jmin]*
 - Ne pas oublier de mettre à jour *Jdroite* et *Tref*
 - Si *Drapeau* Alors
 - On met *Drapeau* à faux
 - Insérez une séquence de déplacements "L" allant de *C->jmin* à *JdroiteSav* dans la solution *S* à la suite du pointeur *Tref* pour la case *C->jmin*
 - [Attention, il faut insérer cette séquence après l'action "S" et non avant]*
 - Ne pas oublier de mettre à jour *Tref*
 - FinSi
 - FinPour
- OUTPUT : une séquence *S* optimale de déplacement du robot

On peut effectuer les remarques suivantes sur cet algorithme :

- toutes les insertions de séquence dans *S* se font avant une action "S" (sauf celle indiquée à la fin de la boucle).
- l'algorithme démarre avec une solution *S* vide et insère un à un des séquences de déplacements correspondant à chaque circuit
- si un circuit est plus à droite de *Jdroite*, on interrompt un déplacement du robot à la case *Jdroite* pour le déplacer jusqu'à ce circuit et effectuer ce circuit, puis on remet le robot à sa place pour poursuivre la séquence.
- deux circuits peuvent être "emmêlés", c'est-à-dire qu'un circuit a son *jmin* entre le *jmin* et le *jmax* de l'autre : dans ce cas on interrompt le premier circuit pour faire l'autre en entier, puis on reprend le premier.

En illustration de cet algorithme, considérons l'instance [1 10 10 0] :



Cette instance contient deux circuits emmêlés :

- **circuit 1** : $jmin=0$ $jmax=6$: Liste : (0,0) (0,6)

- **circuit 2** : $jmin=1$ $jmax=9$: Liste : (0,1) (0,4) (0,5) (0,7) (0,9) (0,8) (0,2) (0,3)

La première itération de la boucle de l'algorithme propose une séquence de déplacement correspondant au circuit 1 : **S R R R R R R S L L L L L S** (12 pas)

Cette séquence prend la pièce en (0,0) puis se déplace en (0,6), réalise un swap "S", puis revient en (0,0) pour déposer la pièce.

A l'issue de cette itération, $Jdroite = 6$ car le robot est allé au plus loin en (0,6).

Les deux circuits sont emmêlés : donc la séquence précédente passe par la case $jmin = 1$ de circuit 2 (c'est-à-dire (0,1)). En fait le robot passe deux fois par cette case : on interrompt alors la séquence précédente en (0,1) lors de son dernier passage, pour y glisser en entier la séquence de circuit 2. On obtient alors la séquence complète finale :

S R R R R R R S L L L L L S R R R S R S R R S L S L L L L L S R S L L S L S (30 pas)

Les fonctions listées dans la fin de cet énoncé ont pour but de vous aider dans l'implémentation de cet algorithme. Elles ne sont là qu'à titre indicatif (vous pouvez choisir votre implémentation). De plus, il est préférable de puiser dans la liste suivante une fonction, de la coder sur un cas simple et de tester immédiatement si ce cas fonctionne bien, cela permet d'avancer pas à pas.

Voici une liste d'instances test de difficulté croissante :

- l'instance [1 10 10 2] ne contient qu'un seul circuit démarrant en case 0
- l'instance [1 6 6 8] ne contient qu'un seul circuit ne démarrant pas en case 0
- l'instance [1 6 6 45] contient deux circuits non emmêlés
- l'instance [1 10 10 0] ne contient que deux circuits emmêlés
- l'instance [1 10 10 30] contient 3 circuits dont les deux premiers ne sont pas emmêlés (mais les deux suivants si).

Q 6.2 Aidez-vous des fonctions suivantes :

- fonction `Cell_char* Ajout_action_apres_c(Solution *S, Cell_char *c, int j, char a, Cell_char **Tref)` qui insère une action "L", "R" ou "S" dans une solution S : soit après le pointeur c d'une case de S si c n'est pas NULL, soit en tête de liste S si c est NULL. La fonction retourne le pointeur sur la cellule créée et insérée dans S . La fonction met à jour également le tableau $Tref$. N'oubliez pas d'incrémenter le nombre de pas.

- fonction `Cell_char* pluscourtchemin_apres_c(Solution *S, Cell_char *c, int j, int l, Cell_char **Tref)` qui insère un chemin fait de "R" ou de "L" dans la solution S entre la case j et la case l du vecteur. [Noter qu'elle revient à itérer correctement la fonction `Ajout_action_apres_c`]. Elle retourne un pointeur sur la dernière cellule ajoutée à S .

- fonction `void Ajout_circuit_dans_solution(LDC *L, Solution *S, Cell_char *c, Cell_char ** Tref, int* Jdroite)` qui insère une séquence de déplacements "L", "R" et "S" du robot dans S correspondant au circuit C . Cette fonction a pour pré-requis que le tableau $Tref$ pour la case $C \rightarrow jmin$ pointe sur un déplacement menant le robot à cette case. Cette fonction consiste à appeler la fonction `pluscourtchemin_apres_c` pour chaque arc de C . La fonction doit mettre à jour en continu $Tref$ et $Jdroite$.

Q 6.3 Construisez une fonction `void algorithme_circuit_CasLigne1x1(Grille *G, Solution *S)` correspondant à l'algorithme de Daniel Graf.

Exercice 7 – TME 10 (semaines 30/04/18 et 07/05/18) - A vous de jouer !

Q 7.1 En utilisant les briques fournies par l'utilisation du graphe et des circuits de permutation, proposer au moins une nouvelle méthode soit pour le cas d'une grille avec "une case par couleur", soit dans le cas général plus difficile d'une grille avec un nombre quelconque de couleurs.

BONUS : Les binômes ayant obtenu les meilleurs résultats se verront attribuer des points bonus ! Il y aura une compétition :

- grille avec "une case par couleur",
- grille avec nombre de couleurs quelconque.

RENDU FINAL TME 11 (semaine du 14/05/18) :

Vous devez rendre l'intégralité du travail (y compris ce qui a déjà été rendu pour la partie 1) en semaine 11.

Pour cela, vous enverrez à votre chargé de TD un fichier .tar ou .zip contenant les fichiers sources ; un rapport les résultats obtenus sur les exercices des DEUX PARTIES ; et les résultats obtenus avec vos algorithmes d'optimisation de la partie 2. Le rapport contiendra également les courbes que vous avez obtenues lors des exercices d'évaluation de la partie 1.

Veuillez vous reporter au document concernant les rendus sur le site.

Votre code doit pouvoir être testé facilement (arguments en ligne de commande ou menu en console).

Vous devez envoyer vos fichiers **le jour précédent la 11ème séance de TME**, avant minuit. A l'occasion de la 11ème séance de TME, vous présenterez votre travail à votre chargé de TD/TME lors d'une mini-soutenance de quelques minutes. A cette occasion, vous lui présenterez notamment les résultats obtenus pour l'exercice 8.

Remarque : En attendant votre tour de passage, vous pouvez mettre en œuvre les exercices du TD11.

Annexe

Preuve de validité et d'optimalité de l'algorithme de Daniel Graf pour le cas d'un vecteur avec “une case par couleur”

On s'intéresse tout d'abord à déterminer des bornes inférieures au problème. En l'occurrence il s'agit de rechercher combien de pas il y a au minimum dans une solution : cela revient à déterminer quels sont les pas essentiels du robot, c'est-à-dire des pas que doit faire le robot quelque soit la solution.

Pour chaque arc $((i, j), (k, l))$ du graphe H , notons d_{ijkl} le plus petit nombre de pas permettant de déplacer le robot entre les deux cases (pour une grille, on sait qu'il s'agit de la valeur $|i - k| + |j - l|$). Dans le cas particulier d'une grille avec “une case par couleur”, le robot doit au moins faire les déplacements correspondant à chaque arc afin de placer toutes les pièces (il doit peut-être effectuer d'autres déplacements en plus). Cette remarque conduit au lemme suivant.

Lemme 1 *Dans le cas d'une grille G avec “une case par couleur”,*

$$d(G) = \sum_{((i,j),(k,l)) \in H} d_{ijkl} \quad \text{est une borne inférieure sur le nombre de pas que doit faire le robot.}$$

On peut remarquer que si H est composé d'un seul circuit commençant en $(0, 0)$, cette borne est atteinte par la solution minimale qui correspond à effectuer les déplacements induits par ce circuit. Dans le cas d'un circuit ne commençant pas en $(0, 0)$, une solution minimale contient un chemin pour aller de $(0, 0)$ à la case du circuit la plus proche puis effectuer les $d(G)$ pas correspondant au circuit puis revenir en $(0, 0)$.

Dans le cas d'un vecteur avec “une case par couleur”, les circuits de H peuvent être triés de manière à ce que l'on connaisse pour chaque circuit C la case $C.jmin$ la plus à gauche du circuit et la case $C.jmax$ la plus à droite du circuit. Un circuit est donné par une liste $C.L$ de déplacements du robot correspondant à la séquence de déplacements des pièces le long du circuit et commençant par la case $C.jmin$. On suppose de plus que la liste LC des circuits est triée selon les valeurs $jmin$.

On construit alors (théoriquement uniquement) un graphe non-orienté \bar{H} à partir de H tel que

- les sommets sont les circuits de H
- il y a une arête entre deux sommets C_1 et C_2 si les deux circuits sont emmêlés, c'est-à-dire si les intervalles $[C_1.jmin, C_1.jmax]$ et $[C_2.jmin, C_2.jmax]$ s'intersectent.

Un graphe non-orienté peut être partitionné en *composantes connexes*, c'est-à-dire en sous-graphes tels que chaque sous-graphe est connexe et que deux sous-graphes ne sont pas reliés par une arête. Notons (D_1, \dots, D_p) les composantes connexes de \bar{H} . Notons pour chaque composante connexe D , $l(D) = \min_{C \in D} C.jmin$ et $r(D) = \max_{C \in D} C.jmax$ c'est-à-dire la case la plus à gauche (resp. à droite) atteint par un circuit de D .

Enfin trions les composantes connexes selon les valeurs $l(D)$.

On a alors le lemme suivant.

Lemme 2 *Pour un vecteur G avec “une case par couleur”,*

$$d(G) + 2 \cdot \left(l(D_1) - 1 + \sum_{i=1}^{p-1} (l(D_{i+1}) - r(D_i)) \right)$$

est une borne inférieure sur le nombre de pas que doit faire le robot.

Ce résultat provient du fait qu’entre deux composantes connexes successives, il faut relier la dernière case à la suivante en déplaçant une fois le robot à l’aller et une fois au retour. (Une preuve plus formelle peut être lue dans l’article de Daniel Graf).

Enfin, on peut prouver que l’algorithme fournit une solution ayant exactement le nombre de pas du lemme précédant, ce qui prouve à l’optimalité de l’algorithme. Pour prouver sa validité, il est possible de rédiger une preuve par récurrence sur le nombre de circuits dans H . En fait à chaque ajout d’un circuit par la boucle de l’algorithme, on construit une sous-solution valide et à la fin toutes les pièces sont correctement déplacées.

En terme de complexité, on peut prouver que l’ajout d’un circuit est linéaire sur le nombre de pas que l’on ajoute. Donc l’algorithme a une complexité de valeur du nombre de pas de la solution fournie, c’est-à-dire en $\theta(n^2)$.

Une preuve plus formelle proposée par Daniel Graf permet d’établir le résultat final suivant.

Théorème 1 *L’algorithme fournit une solution optimale en $\theta(n^2)$.*