Projet C avancé

Outil de visualisation de graphes

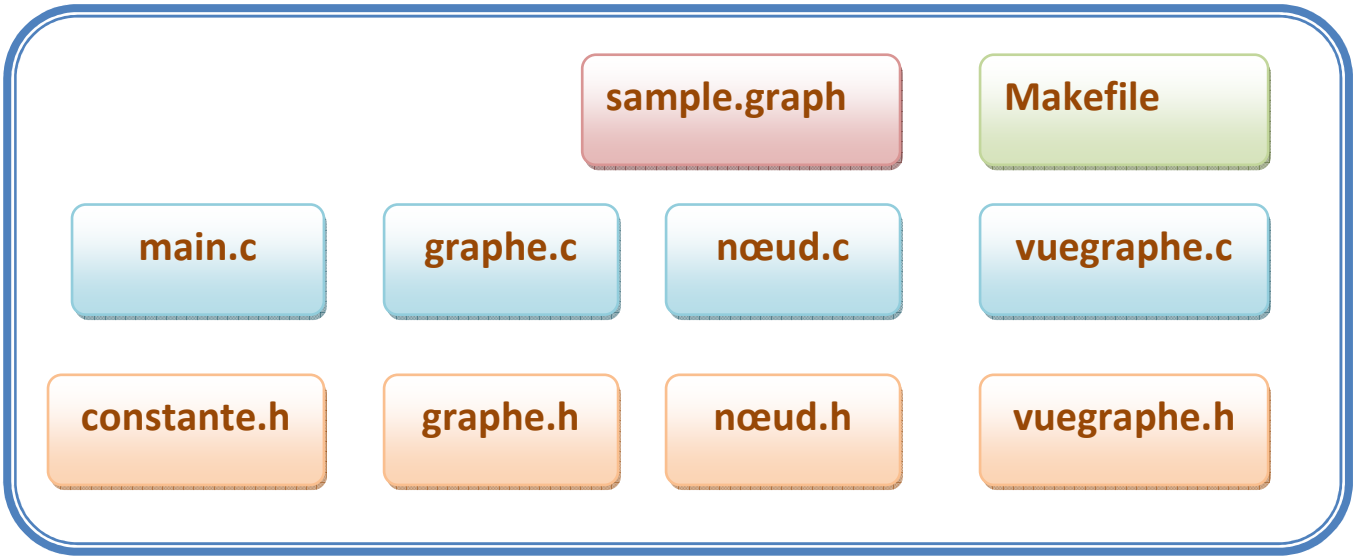
Développement d'une application permettant de trouver les composantes connexes d'un graphe et de les visualiser en les colorants d'une même couleur.

Par Omar EDDASSER – L3 ISC parcours MIAGE
Sous l'enseignement de : M. Amine BOUMAZA
2011 / 2012

Sommaire

I. Structure du projet

1. Les principaux fichiers



sample.graph : Il s'agit dans ce cas du fichier d'entrée contenant l'ensemble des nœuds du graphe. Le chemin du fichier doit être spécifié en argument lors du lancement du programme (ici en utilisant tout simplement le nom du fichier : **sample.graph**)

main.c : C'est le fichier principal de l'application, il permet de lancer celle-ci.

graphe.c : Fichier contenant toutes les fonctions relatives à un graphe.

nœud.c : Fichier contenant toutes les fonctions relatives à un nœud.

vuegraphe.c : Fichier permettant d'afficher graphiquement (en utilisant la librairie SDL) un graphe.

constante.h : Fichier contenant les principales constantes liée à
- l'application (exécution correct ou non d'une fonction…),
- l'interface graphique

Makefile : Fichier permettant de simplifier la compilation du projet

graphe.h, nœud.h et vuegraphe.h sont respectivement les fichiers contenant les prototypes des fonctions, include et structure de données des fichiers : graphe.c, nœud.c et vuegraphe.c

D'autres fichiers sont également présent tel que :
- icon.png : icône de l'application
- angelina.ttf : utilisé dans l'interface graphique (pour afficher le texte : valeur des nœuds)
- *.dll : fichiers utilisés pour faire fonctionner la librairie SDL

2. Description des choix de structures et des fonctions utilisées

a) Main

Le fichier **main.c** contient la fonction du même nom qui :

- Créer un graphe (fonction **initialiserGraphe**)
- Le construit avec le chemin de fichier en argument (fonction **initialiserNoeudDuGraphe**)
- Lance le processus de recherche des composantes connexes (fonction **rechercherComposanteConnexe**)
- Affiche le graphe (fonctions **afficherGraphe** et **afficherGrapheInterface**, la première affiche le graphe sans interface graphique avec seulement des **printf**, c'est celle que j'ai utilisé avant de mettre en place l'interface graphique)
- Libère la mémoire (fonction **freeGraphe**)

b) Graphe

```
struct Graphe
{
	struct Noeud * tab_noeud;
	int nb_noeud;
};
typedef struct Graphe Graphe;
```

Un graphe est une structure composé :

- d'un tableau de nœud
- d'un entier représentant le nombre de nœud dans le tableau

Les fonctions liées au graphe :

```
int initialiserGraphe(Graphe ** graphe);
int initialiserNoeudDuGraphe(Graphe ** graphe, char * filepath);
int initialiserLienNoeudDuGraphe(Graphe ** graphe, char * filepath);
Noeud * chercherNoeud(Graphe ** graphe, int val);
int ajouterNoeudSiInexistant(Graphe ** graphe, int val);
int ajouterLien(Graphe ** graphe, int val, int val2);
int rechercherComposanteConnexe(Graphe ** graphe);
int freeGraphe(Graphe * graphe);
int afficherGraphe(Graphe * graphe);

int getXMax(Graphe * graphe);
int getYMax(Graphe * graphe);
```

initialiserGraphe : Cette fonction s'occupe d'alloué l'espace mémoire du graphe et initialise le nombre de nœud à 0.

initialiserNoeudDuGraphe : Cette fonction remplir le graphe (donné en paramètre) avec le fichier d'entrée (dont le chemin est spécifié en paramètre). Elle ne fait que créer tous les nœuds du graphe.

Algorithme simplifié :

```
ouvrir fichier(chemin)
c ⟵ lireCaractere(fichier)
est_premier_nombre = vrai
tantque (c != findefichier) faire
    si (c != ‘ ’ et c != ‘\n’) alors
        si est_premier_nombre alors
            mot_lu += c
        fsi
    sinon
        si est_premier_nombre alors
            nouveauNoeud(mot_lu)
          mot_lu = ‘’
        fsi
       est_premier_nombre = faux
       si (c == ‘\n’) alors
          est_premier_nombre = vrai
       fsi
    fsi
ftantque
```

La fonction **nouveauNoeud** de l’algorithme correspond à la fonction **ajouterNoeudSiInexistant**. A la fin, de l’exécution de cette fonction, on créer ensuite les liens entre tous les nœuds. (en faisant appel à la fonction **initialiserLienNoeudDuGraphe**).

initialiserLienNoeudDuGraphe : Cette fonction, à la manière de **initialiserNoeudDuGraphe**, lit le fichier, puis ajoute les liens entre les nœuds. En fait, elle ajoute à tous les nœuds un pointeur vers chacun de ses nœuds voisins (un nœud ayant un tableau de pointeur vers ses nœuds voisins, cf. ci-dessous partie Nœud).

chercherNoeud : Cette fonction parcours le tableau de nœud du graphe et vérifie pour chacun d’eux si la valeur de celui-ci est égale à celle qu’elle à reçu en paramètre. Elle retourne soit un pointeur vers le nœud recherché, soit NULL.

ajouterNoeudSiInexistant : Cette fonction créer un nœud s’il n’est pas déjà présent dans le tableau de nœud du graphe (appel la fonction **chercherNoeud** pour savoir si le nœud existe ou pas)

ajouterLien : A partir de deux pointeurs de nœuds, cette fonction ajoute au tableau de pointeur de noeud du premier un pointeur vers le second nœud.

rechercherComposanteConnexe : C’est cette fonction qui recherche les composantes connexes du nœuds. Elle parcours le tableau de nœud du graphe, puis pour chaque nœud :

- s’il est déjà affecté à une composante connexe, ne fait rien
- sinon lui affecte un identifiant de composante connexe, le nœud à son tour affecte à ses nœuds voisins cet identifiant de composant connexe (par récursivité, en parcourant le tableau de pointeur de nœud). Puis incrémente l’identifiant de composante connexe (pour le prochain nœud sans affectation).

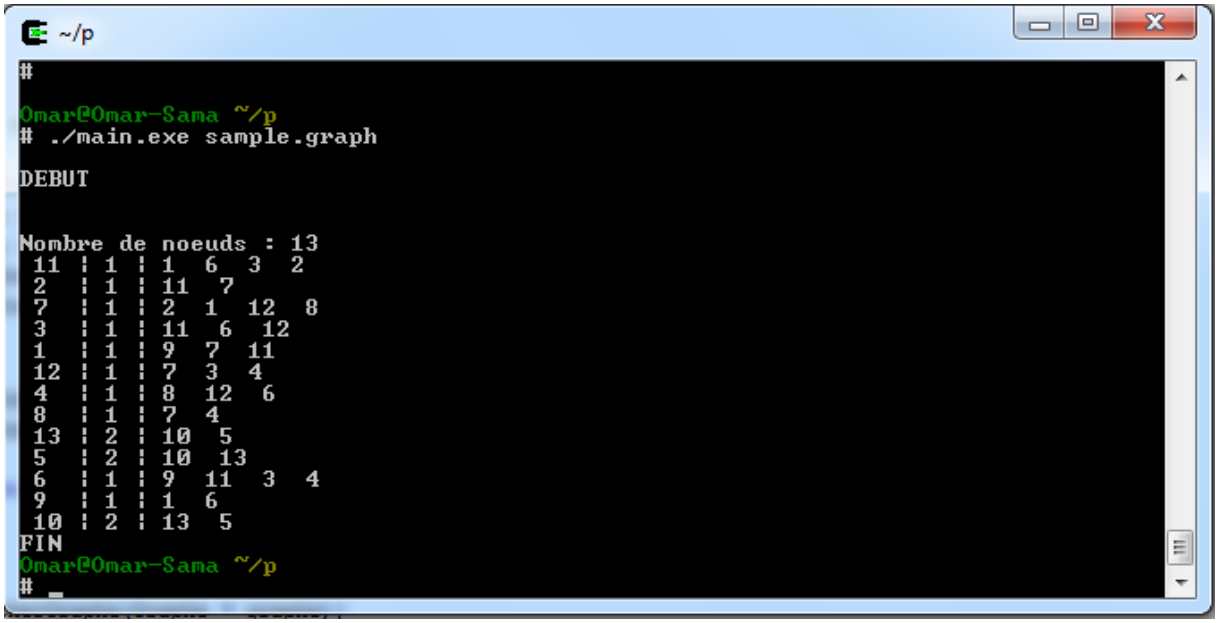
Algorithme simplifié :

```
id_composante_connexe = 1
pour i de 0 à nb_noeud faire
    si nœud[i].id_composante_connexe = -1 alors
        setComposanteConnexe(nœud[i], id_composante_connexe)
        id_composante_connexe++
    fsi
fpour
```

La fonction **setComposanteConnexeNoeud** est abordée un peu plus en détail dans la partie suivante sur les nœuds.

freeGraphe : Cette fonction libère la mémoire occupée par le graphe, elle parcours le tableau de nœud et fait des appels à la fonction **freeNoeud** pour libère l'espace occupé par les nœuds (avant de faire de même avec le tableau de nœud du graphe).

afficherGraphe : Cette fonction affiche le graphe en utilisant des **printf**, elle parcours le tableau de nœud et avec chacun des nœuds fait appel à la fonction **afficherNoeud** (qui exécute la même chose, mais pour les nœuds).
Ci-dessous une capture d'écran de l'affichage du graphe test, avec pour chaque nœud :
sa valeur | l'identifiant de la composante connexe à laquelle il appartient | les valeurs de ses nœuds voisins



getXMax et *getYMax* : Ce sont des fonctions utilisé dans le cadre de l'interface graphique. Elles me permettent notamment de créer une fenêtre suffisamment grande pour que tous les nœuds apparaissent.
Chaque nœud dispose de coordonnées x et y (initialisé à 0) qui sont mise à jours lors de la recherche des composantes connexes du graphe. Ces deux méthodes parcours donc le tableau de nœud du graphe, et retourne la valeur maximale trouvée.

c) Noeud

```
struct Noeud
{
	int valeur;
	struct Noeud ** tab_noeuds_voisin;
	int nb_noeud_voisin;
	int composante_connexe;

	int x;
	int y;
};
typedef struct Noeud Noeud;
```

Un nœud est une structure composé :

- D'une valeur
- D'un tableau de pointeur vers les nœuds voisins
- D'un entier représentant le nombre de pointeur dans le tableau
- D'un identifiant de la composante connexe à laquelle le nœud appartient
- D'un entier x représentant l'abscisse du nœud (utilisé pour l'interface graphique)
- D'un entier y représentant l'ordonnée du nœud (utilisé pour l'interface graphique)

Les fonctions liées au nœud :

```
int afficherNoeud(Noeud * noeud);
int addNoeudVoisin(Noeud * n1, Noeud * n2);
int setComposanteConnexeNoeud(Noeud * noeud, int id_composante, int x, int y);
int freeNoeud(Noeud * noeud);
```

afficherNoeud : Cette fonction à la manière de **afficherGraphe**, affiche un nœud en utilisant des **printf**.

addNoeudVoisin : Cette fonction ajoute au tableau de pointeur du premier nœud, l'adresse du second nœud. Elle créer ou agrandie le tableau de pointeur (**malloc** ou **realloc** en fonction qu'il y a ou non déjà des voisins).

setComposanteConnexeNoeud : Cette fonction va mettre à jour la composante connexe du nœud courant, et s'exécute récursivement avec les nœuds voisins du nœud courant. Elle vérifie d'abord si le nœud courant n'est pas déjà affecté à une composante connexe (pour éviter une boucle infinie puisque deux nœuds sont forcément voisins l'un pour l'autre) et dans ce cas l'affecte de la composante connexe et fait l'appel récursif sur les nœuds voisins.

Algorithme simplifié :

```
si nœud.id_composante_connexe == -1 alors
    nœud.id_composante_connexe = id_comp
    pour i de 0 a nb_noeud_voisin faire
        setComposanteConnexe(nœud_voisin[i], id_comp) // l'appel récursif
    fpour
fsi
```

freeNoeud: Cette méthode tout comme **freeGraphe**, libère la mémoire utilisée par le nœud.

d) Vuegraphe

Les fonctions de vuegraphe:

```
int afficherGrapheInterface(Graphe * graphe);
void afficherNoeudInterface(SDL_Surface * ecran, TTF_Font * police, Noeud * noeud, int nb);
void pause();

void set_pixel(SDL_Surface * surface, int x, int y, Uint32 pixel);
void draw_circle(SDL_Surface * surface, int cx, int cy, int radius, Uint32 pixel);
void fill_circle(SDL_Surface * surface, int cx, int cy, int radius, Uint32 pixel);
void setPixel(SDL_Surface * ecran, int x, int y, Uint32 coul);
void setPixelVerif(SDL_Surface * ecran, int x, int y, Uint32 coul);
void echangerEntiers(int* x, int* y);
void draw_ligne(SDL_Surface * ecran, int x1, int y1, int x2, int y2, Uint32 coul);
```

afficherGrapheInterface : C'est la fonction principale de **vuegraphe**. C'est elle qui affiche un graphe en faisant appel aux autres fonctions du fichier.

- Elle initialise SDL et démarre SDL_TTF (pour l'affichage du texte)
- Ouvre une fenêtre (en utilisant les fonctions **getXMax** et **getYMax** du graphe pour la hauteur et la largeur de la fenêtre)
- Colorie le fond de la fenêtre
- Parcours les nœuds du graphe et les affiches (fonction **afficherNoeudInterface**)
- Puis lorsque l'utilisateur ferme la fenêtre, arrête SDL et SDL_TTF (et libère la mémoire)

afficherNoeudInterface : Cette fonction affiche un nœud dans la fenêtre (donnée en paramètre). Elle dessine d'abord le cercle (de fond), puis affiche la valeur et pour finir affiche les liens (vers les nœuds voisins).

pause : Cette fonction permet simplement de maintenir la fenêtre ouverte, en effet, si l'on ne fait pas appel à cette fonction, la fenêtre apparait (et affiche le graphe) puis disparait aussitôt. J'ai repris cette fonction du site du zéro : http://www.siteduzero.com/tutoriel-3-14090-creation-d-une-fenetre-et-de-surfaces.html et de developpez http://loka.developpez.com/tutoriel/sdl/

Le reste des fonctions de ce fichier (tout comme pour pause()) sont des fonctions que j'ai +/- repris et adapté pour le projet que je ne détaillerais donc pas volontairement dans ce rapport. Il s'agit en effet de simple fonction de base de dessins (*draw_circle :* pour afficher un cercle, *draw_ligne :* pour afficher une ligne, …)

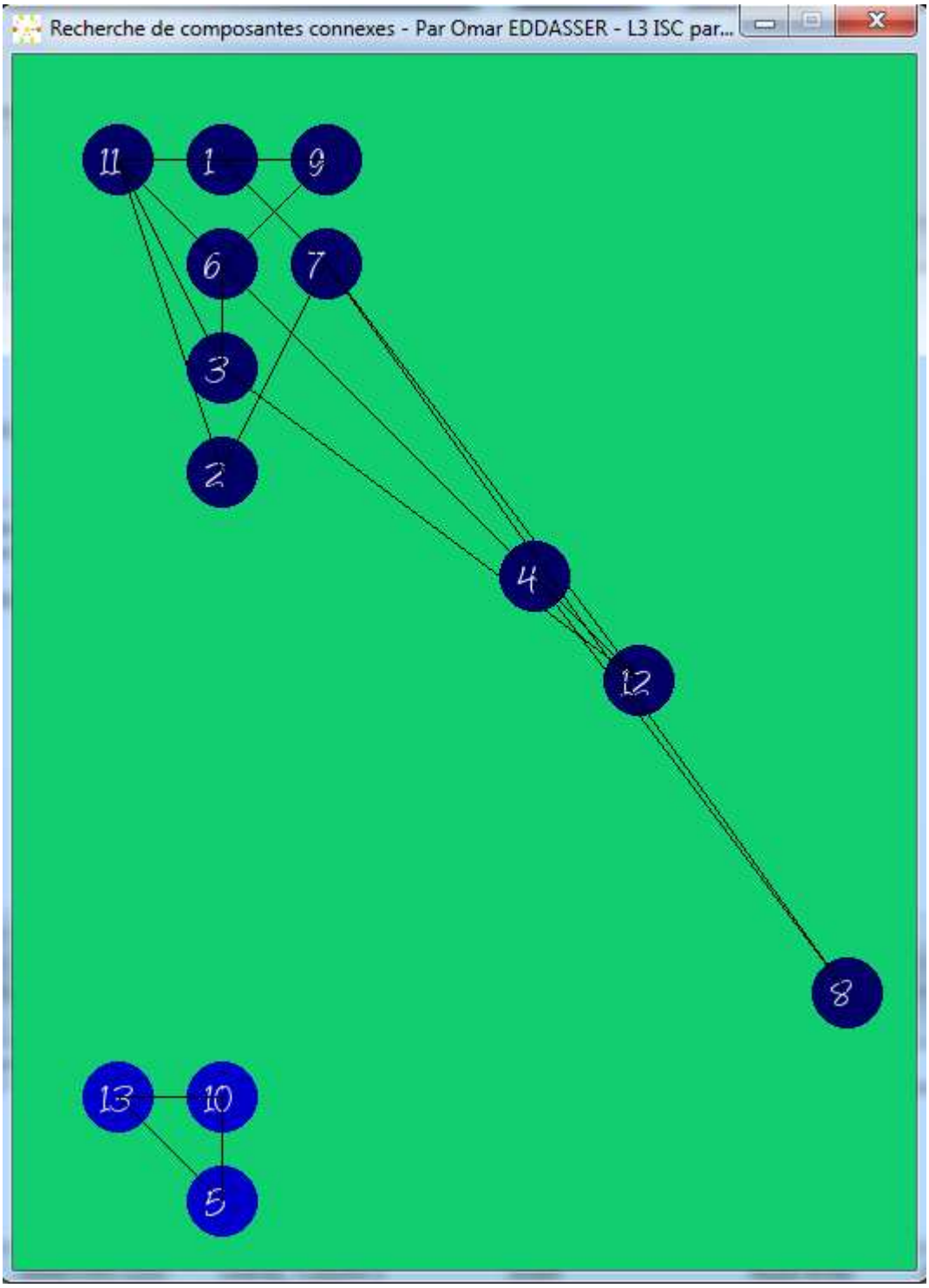
e) Makefile

```
OBJS = main.o graphe.o noeud.o vuegraphe.o
CC = gcc
CFLAGS = -Wall -ansi -g
main.exe : graphe.o noeud.o main.o vuegraphe.o
	$(CC) $(OBJS) -o main.exe

main.o : main.c
	$(CC) $(CFLAGS) -c main.c -o main.o

graphe.o : graphe.c
	$(CC) $(CFLAGS) -c graphe.c -o graphe.o

noeud.o : noeud.c
	$(CC) $(CFLAGS) -c noeud.c -o noeud.o

vuegraphe.o : vuegraphe.c
	$(CC) $(CFLAGS) -c vuegraphe.c -o vuegraphe.o

clean:
	rm *.o *.exe*
```

Le fichier Makefile est utilisé dans mon projet pour faciliter la compilation (la commande make ne compile que les fichiers qui ont été modifié). Elle permet de spécifier un ordre d'exécution en fonction des dépendances des fichiers, dans mon cas fin d'aboutir à un exécutable (main.exe), je dois avoir un graphe.o, nœud.o, main.o et vuegraphe.o. Chacun de ces fichiers étant lui même obtenu par une compilation d'autres fichiers (noeud.o est obtenu grâce à noeud.c …)

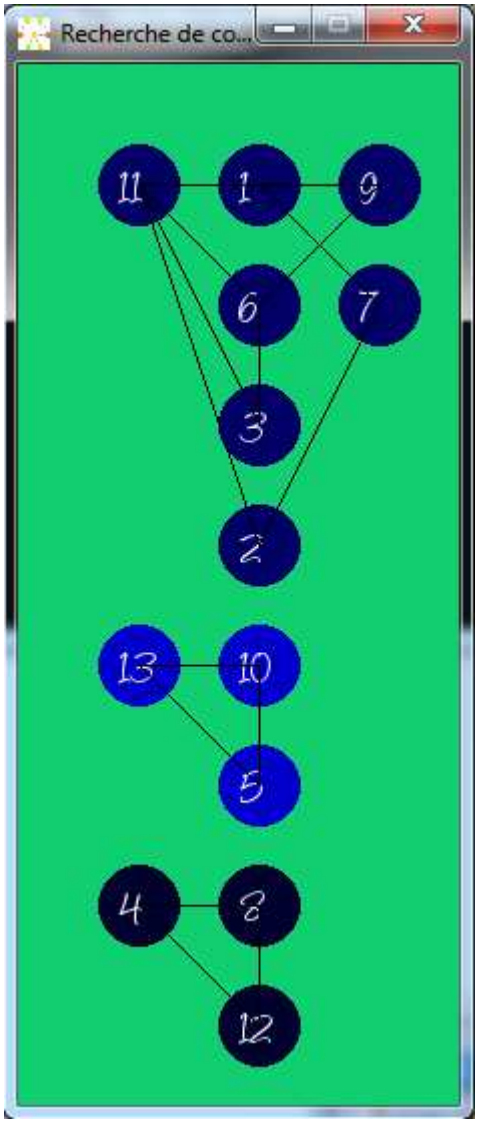
II. Captures d'écran

Capture d'écran avec le jeu test :



11 1 6 3 2
2 11 7
7 2 1 12 8
3 11 6 12
1 9 7 11
12 7 3 4
4 8 12 6
8 7 4
13 10 5
5 10 13
6 9 11 3 4
9 1 6
10 13 5

Capture d'écran avec un autre jeu test (j'ai simplement crée une autre composante connexe avec les nœuds 4, 8 et 12, le reste des nœuds étant inchangés) :



11 1 6 3 2
2 11 7
7 2 1
3 11 6
1 9 7 11
13 10 5
5 10 13
6 9 11 3
9 1 6
10 13 5
4 8 12
12 4 8
8 12 4

III. Notes sur les choix

Le fichier est lu deux fois :

- Une première fois par la fonction **initialiserNoeudDuGraphe** qui va créer tous les nœuds du graphe
- Une seconde fois par la fonction **initialiserLienNoeudDuGraphe** qui va ajouter à chaque nœud un pointeur vers chacun de ses nœuds voisins.

Cela s'explique par le fait que l'on ne connait pas à l'avance le nombre de nœud du graphe (nombre de ligne du fichier), et donc que je réalloue un espace supplémentaire dans le tableau de nœud du graphe à chaque fois que je rencontre un nouveau nœud. La fonction **realloc** permet bien de réallouée un espace supplémentaire, cependant en utilisant cette fonction, l'adresse du tableau (et donc des nœuds contenu dans ce tableau) change, or j'utilise dans chaque nœud un tableau de pointeur vers chacun des nœuds voisins. Ce qui fait que lorsque je construis un premier nœud, si je lui ajoute les adresses de ses nœuds voisins, puis si j'utilise la fonction **realloc** pour ajouter un nouveau nœud au

tableau, les adresses de tous les nœuds change, mais dans la structure nœud le tableau de pointeur, pointe toujours vers les anciennes adresses des nœuds voisins. J'ai donc besoin de créer d'abord tous les nœuds, de façon à ce qu'il ait tous une adresse (celle-ci change à chaque fois que je refais un **realloc**, mais peut importe donc) définitive. Puis ensuite, je parcours le fichier une seconde fois, (sans avoir à créer de nœud, donc les adresses mémoire ne change plus) et j'ajoute pour chacun d'eux l'adresse de ses nœuds voisins.

Il aurait été possible de ne parcourir le fichier qu'une seule fois, en stockant par exemple la valeur du nœud voisin dans le tableau du nœud, mais j'ai souhaité sauvegarder l'adresse des nœuds voisins plutôt que la valeur, car c'est plus simple après avec mon algorithme pour rechercher les composantes connexes.

Le fichier est lu caractère par caractère, car dans le sujet il est préciser de ne pas faire d'hypothèse sur la taille d'un graphe, par conséquent, le nombre de nœud voisins pour un nœud quelconque est inconnu et donc la taille d'une ligne d'un fichier est aussi inconnue.