Rapport de projet

Outil de visualisation de graphes

Hurier Médéric
mederic.hurier@etudiant.univ-nancy2.fr
Licence ISC parcours MIAGE

Année 2011-2012

Table des matières

1. Contexte du projet

1.1. Sujet

Le but de ce projet à faire en binômes est de programmer une application qui trouve les composantes connexes dans un graphe et de visualiser le graphe en colorant d'une même couleur les composantes connexes.

Un graphe est un ensemble de nœuds, dont certaines paires sont directement reliées par un lien (voisins). Un chemin dans un graphe est une séquence de nœuds telle qu'il existe un lien entre chaque paire successive de nœuds dans la séquence. Une composante connexe dans un graphe est un ensemble de nœuds connectés de telle sorte qu'il existe un chemin entre tous les nœuds de l'ensemble.

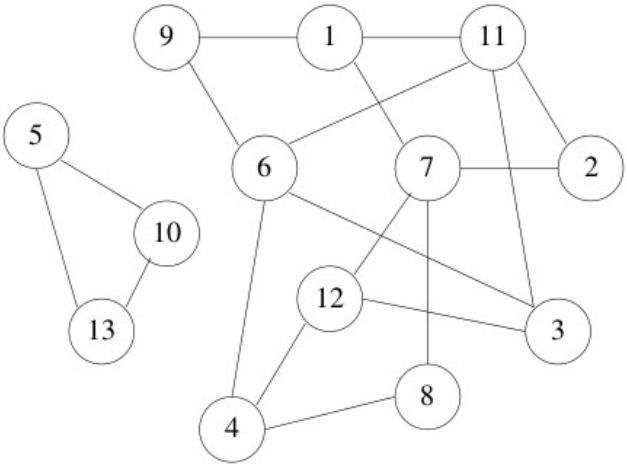


FIGURE 1 – Exemple de graphe composé de 13 noeuds. Les noeuds 5,10 et 13 forment une composante connexe, le reste des noeuds forme l'autre composante connexe. Un exemple de chemin : 4,6,3,12,7,2.

1.2. Choix de développement

La réalisation du projet a nécessité une prise de décision globale concernant les axes de développements et les conventions afin de mieux cadrer la réalisation du projet.

Ainsi, les qualités logicielles retenues pour l'application sont les suivantes:

- **Fiabilité** : gestion robuste de la mémoire et des données en entrée
- **Maintenabilité** : documentations, commentaires et découpage des fonctions
- **Portabilité** : respect de la norme ANSI, aucun avertissements
- **Performance** : utilisation fine de la mémoire pour éviter les redondances

Au détriment des fonctionnalités et de la facilité d'utilisation, ces critères permettent d'assurer la **compréhension** et **l'extensibilité** du projet.

1.3. Technologies et outils

L'utilisation du **langage de programmation C** a été fixée par le cahier des charges comme contrainte de réalisation. La manipulation d'un graphe nécessitant une gestion fine de la mémoire pour éviter son encombrement, c'est une technologie de bas niveau très adaptée, avec sa gestion fine des pointeurs et sa rapidité d'exécution.

Une autre contrainte est l'usage de la **bibliothèque SDL**. Populaire dans les applications multimédia et les jeux vidéos, elle permet de réaliser avec une certaine facilité des dessins simples. Pour améliorer le rendu, une extension nommée **« SDL_ttf »** apporte le support du texte pour le nommage des nœuds.

Ces deux bibliothèques doivent être installées sur le système pour que celle ci puisse s'exécuter normalement. A défaut, des messages d'erreurs seront affichés.

Bien que n'étant imposé par le cahier des charges, un **environnement de développement intégré** a été choisi pour coder l'application. Il permet d'augmenter la **productivité** du développeur, tout en lui fournissant **des outils évolués** tel qu'un débugger et l'auto complétion.

La **portabilité** est assurée par la création d'un fichier **MakeFile**, rendant la compilation et l'édition des liens possibles pour d'autres plateformes et outils.

L'outil qui a été choisi est **Netbeans** sous licence CDDL/GPL, supporté par Oracle et une communauté open source.

2. Étape de conception

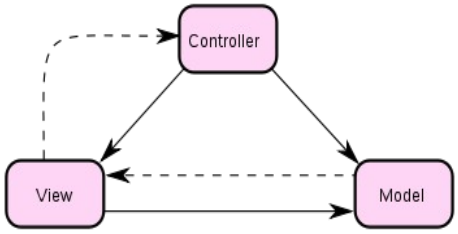
2.1. Choix du modèle MVC

Le modèle MVC est une **méthode de conception** populaire permettant d'organiser une application graphique. Sa finalité est de rendre le code plus découplé et maintenable. L'acronyme signifie « **Modèle-Vue-Contrôleur** ». Chaque élément correspond à une couche dont les rôles sont bien définis :

- Modèle : description des données manipulées et la logique métier
- Vue : présentation des données à l'utilisateur
- Contrôleur : algorithmes de traitement et gestion des événements

Cette méthode est adaptée au projet, car il permet **d'isoler** les algorithmes de calcul de la structure de données et de sa présentation.

Le détail de l'implémentation est développée dans la partie 3.1

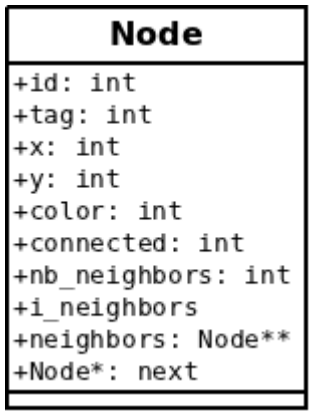


2.2. Structures de données

2.2.1. Structure nœud

Un nœud est un **composant élémentaire** du graphe. A chaque nœud est attribué un **identifiant** permettant de le distinguer de ces paires. Ils peuvent être reliés entre eux, donnant lieu à des **connexions** entre les éléments. Des nœuds connectés sont appelés des **voisins**.
Les nœuds sont destinés à être **représenté graphiquement**. Ainsi, plusieurs informations permettant de décrire leurs **états d'affichage** ont été ajoutées.

Voici un schéma présentant la structure nœud pour l'application :



Les champs de la structure sont :

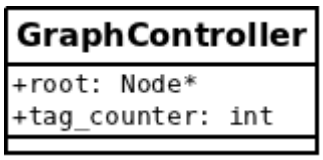
- id : identifiant unique pour le nœud
- tag : identifiant du graphe auquel il est rattaché
- x : coordonnées du point en abscisse
- y : coordonnées du point en ordonnée
- color : code couleur du nœud
- connected : état de connexion du nœud (0 non traité, 1 complètement connecté)
- nb_neighbors : nombre de voisins
- neighbors : tableau de pointeur de nœud. Référence vers les voisins
- next : Élément suivant pour assurer le chaînage entre élément (liste chaînée)

2.2.2. Structure contrôleur

Le contrôleur de l'application regroupe les **traitements** propres à la gestion des graphes. Il permet entre autre **d'analyser** les données en entrée, de **procéder** aux calculs de composantes connexes et de **visualiser** les informations.

Il utilise les structures nœud et visualiseur pour **présenter** et **représenter** l'information.

Les fonctions du contrôleurs sont **bien identifiés** pour améliorer **l'extensibilité**.



Les champs de la structure sont :

- root : premièr élément de la liste chaînée utilisée pour stocker les nœuds
- tag_counter : compteur interne pour stocker le nombre de graphe identifié

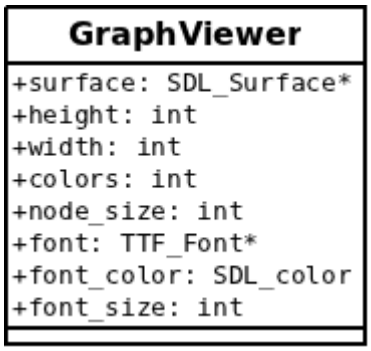
Les principales fonctions utilisées pour cette structure sont :

- parse : analyse un fichier d'entrée au format spécifié par le cahier des charges
- tag : fonction de calcul des graphes connexes. Ajoute un identifiant au nœuds
- print : affiche le contenu du graphe dans la sortie standard
- stats : affiche des statistiques (nombre de nœuds ...)
- display : montre une fenêtre pour visualiser les graphes

2.2.3. Structure visualiseur

Pour apprécier rapidement les différents graphes, l'application ouvre **une fenêtre** avec les **nœuds**, leurs **connexions** et les **graphes** auxquels ils sont associés (code couleur). Le visualiseur utilise SDL comme bibliothèque graphique.

Le positionnement des nœuds et le choix des couleurs sont **aléatoires**. Les algorithmes de placement essaieront **d'éviter les superpositions dans une certaine limite**. Ainsi, ces informations peuvent **variées** entre les exécutions mais permettent un affichage **plus rapide** pour les petits volumes et **plus souple** pour les gros volumes.



Les champs de la structure sont :

- surface : surface de dessin (contenu de la fenêtre)
- height : hauteur de la fenêtre en pixel
- width : largeur de la fenêtre en pixel
- colors : nombre de couleurs (32 bits)
- node_size : taille d'un nœud (diamètre) en pixel
- font : objet police importé à l'initialisation du visualiseur depuis le répertoire data
- font_color : code couleur de la police
- font_size : taille de la police en pixel

2.3. Algorithmes

2.3.1. Analyse du fichier d'entrée

Cet algorithme décrit le **remplissage** des nœuds à partir d'un fichier d'entrée. La première lecture permet de **créer les nœuds** avec une **valeur** et un **nombre de voisin** (allocation). La seconde lecture **ajoute les voisins** à partir de la liste précédemment créée.

```
Fichier f = Ficher(chemin_fichier)
Entier v, id

# Première lecture
Tant que v = valeur_lue(f) :
     nombre_voisin = aller_ligne_suivante(f)
     Noeud n = Noeud(v, nombre_voisin)
     Si racine :
          ajouter_noeud(racine, n)
     Sinon :
          racine = n

rembobiner(f)

# Deuxième lecture
Noeud curseur = racine
Tant que curseur :
     id = valeur_lue(f)
     Tant que NON fin_ligne(f) :
          v = valeur_lue(f)
          Noeud voisin = trouver_noeud(v)
          ajout_voisin(curseur, voisin)
     curseur = curseur.suivant

fermeture(f)
```

Fonction implémentant cet algorithme :

src/controllers/graph_controller.c:gctrl_parse()

2.3.2. Calcul des composantes connexes

Le calcul des composantes connexes relie les nœuds à leurs voisins. Ces liaisons forment un **graphe**, identifié par **un numéro unique (tag)**.
L'algorithme se base sur **un appel récursif** pour parcourir en largeur les voisins.

```
# Tag un nœud et ses voisins récursivement
# Retourne VRAI si un nœud a été tagué, FAUX sinon
Def tag(noeud, tag) :
     Si nœud.tag :
          retourne FAUX

     # Tag le nœud
     nœud.tag = tag

     # Appel récursif
     Pour chaque voisin dans nœud.voisins :
          tag(voisin, tag)

     retourne VRAI
```

```
Entier compteur = 1 # Attribution des tag

Noeud curseur = racine
Tant que curseur :
     Si tag(curseur, compteur :
          compteur += 1

     curseur = curseur.suivant
```

Fonction implémentant cet algorithme :

src/controllers/graph_controller.c:gctrl_tag()

src/models/node.c:node_tag()

2.3.3 Cas de gestion

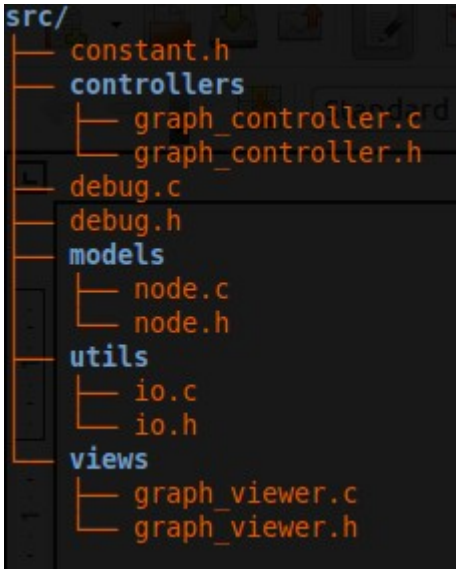
Voici les différentes erreurs possibles et correctement gérées par l'application :

- Ligne vide au début, au milieu ou à la fin du fichier d'entrée
- Voisin non référencé, gestion de la mémoire
- Testé avec plus de 40 valeurs, dont des négatives

3. Étape de réalisation

3.1. Structure de l'application

L'application se base sur un modèle MVC, en suivant **sa hiérarchie des répertoires** et **ses conventions de nommage**. Ainsi, on retrouve dans le dossier source « src » **un dossier par couche** du modèle (Modèle, Vue, Contrôleur) avec des fichiers contenant les structures et les fonctions du module.



Le dossier « utils » est un **module utilitaire** permettant entre autre de procéder aux **accès entrées/sorties** de l'application. Le fichier « constant.h » contient des variables de **configuration**. Un **module de débogag**e, composé des fichiers « debug.h » et « debug.c » gère les messages retournées à l'utilisateur lors de l'exécution.

Dans le dossier du projet, on trouve également un fichier « Makefile » permettant de compiler le projet avec **les outils GNU**. Les binaires générés sont placés dans le dossier « dist » selon la plate forme et le mode de diffusion (production ou debug).

Le dossier « doc » est composé d'un dossier « html » avec **la documentation technique** du projet au format HTML. Pour y accéder, il suffit d'ouvrir le fichier « index.html ».

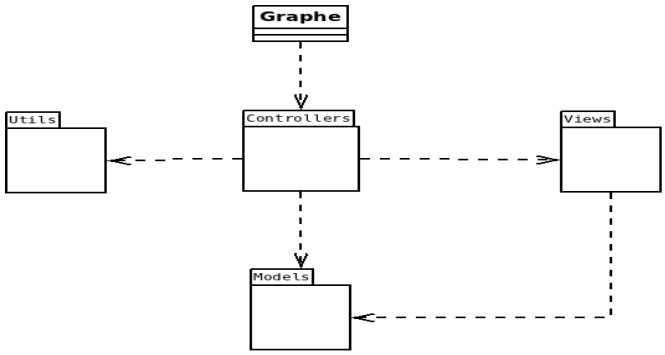
Enfin, le dossier data contient **les ressources** utilisés pour l'application.

- Sample.graph : fichier d'exemple pour l'utilisation du graphe
- ubuntu.tff : police utilisé par l'application

ATTENTION : Le dossier « data » doit être placé prêt du binaire après la compilation. Le cas contraire, un message d'erreur s'affichera lors de l'exécution pour avertir l'utilisateur.

3.2. Dépendances

Comme pour la structure de l'application, les dépendances sont directement **induites** par l'utilisation du modèle MVC.



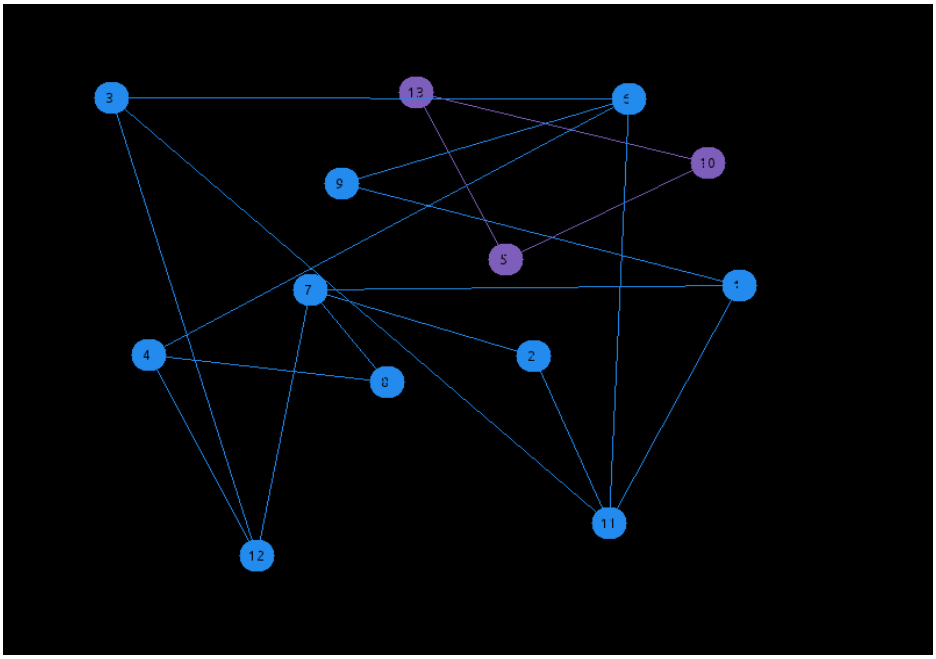
Le point d'entrée du programme est le fichier « graphe.c » qui fait directement appel au fichier du module contrôleur « graph_controller » pour en utiliser les fonctions.

Le contrôleur est **le chef d'orchestre** de l'application. C'est lui qui alimente les structures de données en utilisant les fonctions de lecture du module « utils ».

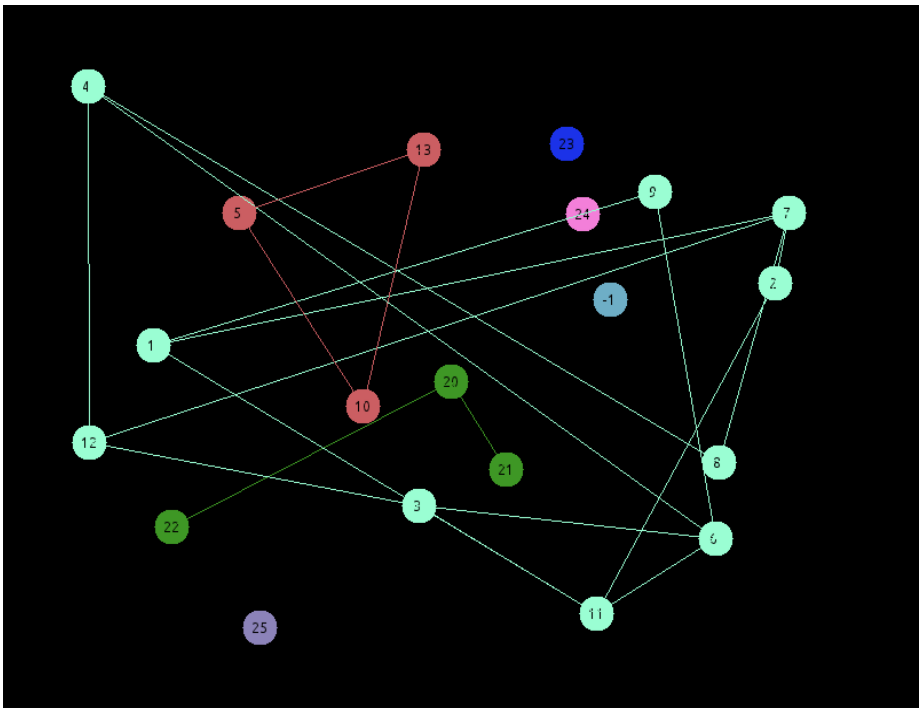
Le module « views » est également utilisé par ce contrôleur. En connaissant le modèle de donnée, il peut les afficher à l'utilisateur.

Enfin, le module « models » contient **la structure principale** de l'application : nœud.

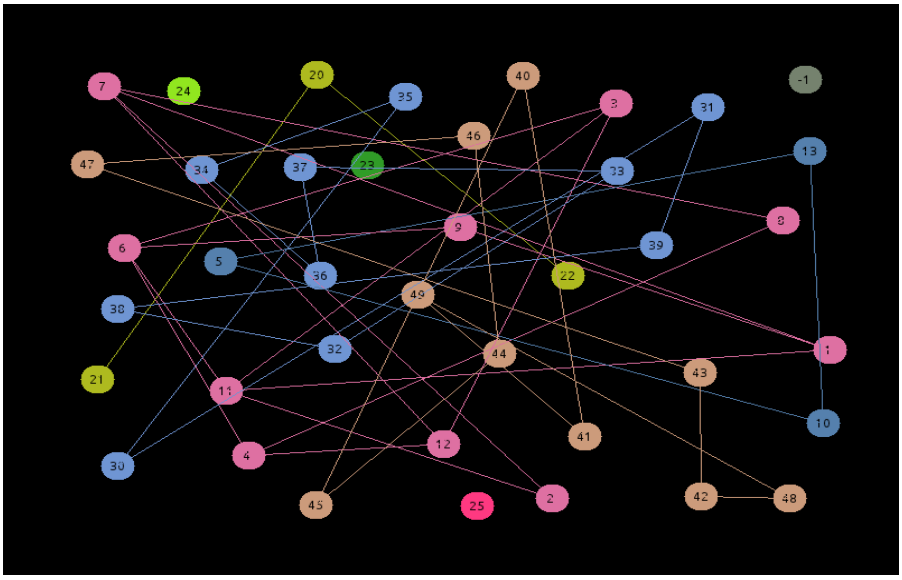
3.2. Exemples de graphes visualisés



Deux graphes à partir des données du cahier des charges



3 graphes avec des voisins, et 4 points sans voisins



Cas avec plus de 40 nœuds, dont des valeurs négatives