Tables de routage

Bouton Nicolas, Dedarally Taariq, Trinh Gia Tâm

Mai 2019

Table des matières

Chapitre 1

Introduction

1.1 Objectif

Le but de ce projet est de créer une application qui calcule la table de routage de chaque noeuds d'un réseaux de 100 noeuds (graphe de 100 noeuds).

1.2 Instruction de compilation

Le programme est écrit en C. Pour compiler, il suffit de taper "make", puis pour exécuter taper "./graph".

1.3 Contenu du projet

Description du contenu dans le projet :
— main.c programme principal
— graph.c/h pour la création du graphe
— connexe.c/h pour le parcours du graphe
— routage.c/h tableau de routage
— const.h contient les constantes utilisées
— affiche.c/h pour afficher le graphe dans une fenetre Canvas
— moteur_graphique.c/h bibliothèque fils de flame11
— flame.c/h bibliothèque basée sur X11 crée par Yaspr
— pi.h outils de la bibliothèque fournit par Yaspr
— Makefile permet de compiler le programme

Chapitre 2

Structure de données

Nous avons décidé d'utiliser des pointeurs vers les structures au lieu de les passer en argument sans pointeurs.

2.1 Structure du Graphe

Cette structure contient 2 champs :
— un tableau qui représente le graphe
— une structure qui permet de bien initialiser le graphe

2.2 Tableau du graphe

Ce tableau représente le graphe. S'il y a une arête entre i et j, alors le poids de l'arête est noté, sinon il y a -1 dans list[i][j] et list[j][i].

2.3 Insert

Cette structure permet de bien initialiser le graphe et contient 2 champs :
— un tableau qui compte le nombre d'arrête vers le même tier
— un tableau où est noté le nombre d'arrête qu'il faut avoir vers le même tier (est utilisé uniquement pour le tier 2 et 3)

2.3.1 Compteur

Ajoute 1 a la valeur au sommet i à chaque fois qu'on ajoute une arête vers le même tier.

2.3.2 Proba

Ce tableau est initialisé au moment où on initialise le graphe et permet de savoir combien d'arêtes il peut y avoir vers un noeuds du même tier. Pour

le tier 2 et 3 lorsque on ajoute une arête entre le sommet i et j il faut aussi l'ajouter de j vers i et sauvegarder qu'il y a une arête vers le même tier.

2.4 Table de routage

Cette structure contient 2 champs :
— un tableau qui représente la table de routage avec le poids
— un tableau qui représente les successeurs

2.4.1 Poids

Ce tableau représente le poids minimum pour aller d'un sommet vers un autre. Par exemple, prenons i le sommet de départ et j le sommet d' arrivé. poids[i][j] indique le poids minimum pour aller de i à j.

2.4.2 Successeur

Ce tableau représente les noeuds à qui il faut envoyé le message sachant le destinataire. Par exemple le sommet i est l' envoyeur, j le destinataire, donc le sommet k = succ[i][j] est le sommet à qui le sommet i doit envoyer le message pour que le chemin entre i et j soit le plus cours.

Chapitre 3

Description du projet

3.1 Création du graphe

3.1.1 Initialisation du graphe

On alloue la mémoire nécéssaire pour créer un graphe. On initialise tout le tableau du graphe à -1 pour dire qu'il n'y a pas d'arêtes pour l'instant. Maintenant, on initialise ces arêtes :

- les Backbones (Tier 1) avec aucune arête entre eux pour l'instant.
- les opérateurs de niveau 2 (Tier2) avec un nombre d'arêtes associés au même niveau qui varie entre 2 et 3 ;
- les opérateurs de niveau 3 (Tier3) avec un nombre d'arêtes associé au même niveau qui vaut 1.

Pour chaque sommet, on initialise à 0 le nombre d'arêtes liés d'un sommet à un autre de même rang.

3.1.2 Calcul du graphe

En parcourant les listes, en vérifiant qu'on ne tombe pas sur la diagonale et qu'il y a une arête, on définit aléatoirement le poids de l'arête :

- Tier1 : entre 5 et 10.
- Tier2 : entre 10 et 20.
- Tier3 : entre 15 et 50.

Et on augmente le compteur du sommet traité à chaque fois qu'on définit un poids d'un lien qui même vers un sommet de même rang.

3.2 Vérification de la connexité

On fait un parcours en profondeur du graphe sur chaque sommet à l'aide de la coloration pour vérifier si le graphe est connexe. On a 3 couleurs.

- 0 = non traité
- 1 = en cours de traitement

— 2 = déjà traité

Un compteur est présent pour indiquer si le graphe est connexe. Initialisé à 0, il augmente de 1 à chaque fois qu'on a fait un parcours. Dans l'idéal, faire ce parcours une seule fois uniquement indique que tous les sommets ont tous été traités du 1er coup et donc que le graphe est connexe.

3.3 Création de la table de routage

Pour la création du chemin on crée une table de routage.La table de routage est composée de deux matrices :une pour les successeurs et une autre pour les poids.On applique ensuite l'algorithme de Floyd-Warshall.

3.4 Reconstitution du chemin

Pour restituer le chemin on prend la table de routage calculée et on cherche le plus court chemin dans le tableau de hashage, fonction des identifiants entrée (l'expéditeur/destinataire).

3.5 Affichage du graphe

3.5.1 Description

L'affichage du jeu se fait sur une fenêtre graphique :les noeuds sont représentés par des cercles et pour afficher le chemin le plus court entre deux noeuds il faut cliquer sur les cercles. Le plus court chemin sera affiché en jaune.Pour quitter proprement la fenêtre graphique il faut appuyer sur 'q' du clavier.

3.5.2 Fonctionnement de l'affichage

La fenêtre graphique est géré par la bibliothèque flame11 crée par Yaspr (lien github en annexe pour consulter le code source) basée sur X11.On a crée un bibliothèque fils (moteur_graphique.c/h) pour plus facilement faire la gestion de l'affichage d'un graphe.

Chapitre 4

Annexe

Lien du code source sur github :

`https://github.com/Sholde/ProjetAlgo/tree/master/src`

Ici dessous les différents fichiers imprimés en pdf :

4.1 Makefile

Fichier de compilation :

```
# Evite conflit:
.PHONY: all lib compil clean

# Constantes:
WARN = -Wall
OPTIM = -g3 -Ofast
GCC = @gcc
MSG = @echo compilation en cours: création de $@
FLAME = -std=c99 -O2 -ffast-math flame.c -lm -lX11
# all
all: graph main.o affiche.o graph.o connexe.o routage.o flame.o moteur_graphique.o clean

# Run:
run:
    @echo -n lancement du programme:
    ./graph
    @echo fin de programme:

# Edition de lien du programme principal
graph: main.o graph.o routage.o connexe.o affiche.o moteur_graphique.o
    ${GCC} -o $@ $^ ${OPTIM} ${WARN} ${FLAME}
    ${MSG}: fichier executable

# Compilation du programme principal
main.o: main.c *.h
    ${GCC} -c ${OPTIM} ${WARN} $*.c
    ${MSG}

# Compilation des fichiers de gestions du programme
affiche.o: affiche.c *.h
    ${GCC} -c ${OPTIM} ${WARN} $*.c
    ${MSG}

graph.o: graph.c *.h
    ${GCC} -c ${OPTIM} ${WARN} $*.c
    ${MSG}

connexe.o: connexe.c *.h
    ${GCC} -c ${OPTIM} ${WARN} $*.c
    ${MSG}

routage.o: routage.c *.h
    ${GCC} -c ${OPTIM} ${WARN} $*.c
    ${MSG}

flame.o: flame.c *.h
    ${GCC} -c ${OPTIM} ${WARN} $*.c
    ${MSG}

moteur_graphique.o: moteur_graphique.c *.h
    ${GCC} -c ${OPTIM} ${WARN} $*.c
    ${MSG}

# Netoyage des fichiers
clean:
    @echo netoyage des données supression de: *.o
```

```
57      @rm *.o
58
```

4.2 Création du graphe

Fichiers pour la création du graphe :

```c
#include "const.h"

/**
 * Algorithme de création de graphe selon les modalités de l'énoncé
 * */

typedef struct {
    int compteur[TAILLE_GRAPHE][3];     // >compte nb voisin vers chaque tier dans la liste
    int proba[TAILLE_GRAPHE][1];        // >nb de noeuds vers le meme tier
} insert;

// Graphe représenté par une liste d'adjacence
typedef struct {
    int list[TAILLE_GRAPHE][TAILLE_GRAPHE];     // >pointe vers 100 listes** qui pointe chacune vers 3 liste* ( qui sont les liste de voisin vers un tier1, tier2, tier3) qui pointe vers une liste
    insert I;                                   // >permet de bien initialiser le graphe
} graphe;

// Génére un graphe
// @return graphe graphe generé
graphe* init_graphe();


// Verifie si le sommet i est dans la liste de j et inversement
//( meme si ca sert a rien car quand on ajoute un voisin a un sommet on
// ajoute aussi le sommet au voisin ) est dans la liste
// @param G le graphe
// @param i sommet
// @param etat_i dans quel tier est le sommet i
// @param j sommet
// @param etat_j dans quel tier est le sommet j
// @return booléens
int verifSiSommetInListe(graphe* G, int i, int j);

// Test si le nb de noeuds max est atteint
// @param G le graphe
// @param i sommet
// @param noeudsMax le noeuds maximum
// @param etat dans quel tier appartient le sommet i
// @return booléens
int test_noeuds_max(graphe* G, int i, int noeudsMax, int etat);

// Calcul quel noeuds choisir
// @param G le graphe
// @param sommet un sommet du graphe
// @param deb debut
// @param fin fin
// @param noeudsMax nb de noeuds max du sommet vers le meme tier
// @param etat_sommet dans quel tier appartient le sommet dans lequelle on cherche un voisin
// @param etat_i dans quel tier appartient le sommet i ( le voisin )
// @return noeuds
int calcul_noeuds(graphe* G, int sommet, int deb, int fin, int noeudsMax, int etat_sommet, int etat_i);
```

```
52
53  // calcul les arc du tier 1
54  // @param G le graphe
55  // @return un graphe géneré
56  void calculTier1(graphe* G);
57
58  // calcul les arc du tier 2
59  // @param G le graphe
60  // @return un graphe géneré
61  void calculTier2(graphe* G);
62
63  // calcul les arc du tier 3
64  // @param G le graphe
65  // @return un graphe géneré
66  void calculTier3(graphe* G);
67
```

```c
#include <stdlib.h>
#include <stdio.h>

//~ local
#include "graph.h"
#include "const.h"

graphe* init_graphe(){
    graphe* G = malloc(sizeof(graphe));

    int i, j, p = 0;

    // struct liste
    for(i = 0; i < TAILLE_GRAPHE; i++) // ini toute le tableau a -1
    {
        for(j = 0; j < TAILLE_GRAPHE; j++)
        {
            G->list[i][j] = -1;
        }
    }

    // struct Insert
    int found = 0;
    int cmp2;
    while(!found)
    {
        cmp2 = 0;
        for(i = 0; i < TAILLE_GRAPHE; i++)
        {
            if( i < nbTier1 )
            {
                G->I.proba[i][0] = 0;
            }
            if( i >= debTier2 && i < finTier2)
            {
                p = rand()%2 + 2;
                G->I.proba[i][0] = p;
            }
            if( i >= debTier3 && i < finTier3)
            {
                G->I.proba[i][0] = 1;
            }
            for(j = 0; j < 3; j++)
            {
                G->I.compteur[i][j] = 0;
            }
        }

        for(i = debTier2; i < finTier2; i++)
        {
            cmp2 += G->I.proba[i][0] = p;
        }

        if( cmp2 % 2 == 0 )
        {
            found = 1;
        }
```

```
58      }
59      return G;
60  }
61
62  int verifSiSommetInListe(graphe* G, int i, int j) {
63
64      if(G->list[i][j]!= -1)
65          return 1;
66      return 0;
67  }
68
69  int test_noeuds_max(graphe* G, int i, int noeudsMax, int etat) {
70
71          if(etat == tier1)
72          {
73              if(G->I.compteur[i][etat] < noeudsMax)
74                  return 1;
75          }
76          else
77          {
78              if(G->I.compteur[i][etat] < noeudsMax &&  G->I.compteur[i][etat] <
                G->I.proba[i][0])
79                  return 1;
80          }
81
82      return 0;
83  }
84
85  int calcul_noeuds(graphe* G, int sommet, int deb, int fin, int noeudsMax, int
    etat_sommet, int etat_i) {
86
87      int distance = fin - deb;
88      int pointeur[distance+1];
89      int i, j = 0;
90
91      for(i = 0; i < distance; i++)
92          pointeur[i] = 0;
93
94      for(i = deb; i < fin; i++)
95      {
96          if(G->I.compteur[i][etat_i] < noeudsMax
97              && !verifSiSommetInListe(G, sommet, i)
98              && i != sommet)
99          {
100             pointeur[j] = i;
101             j++;
102         }
103     }
104     if(j != 0)
105     {
106         int a, b;
107         b = rand();
108         a = b % j;
109         b = pointeur[a];
110         return b;
111     }
112     return -1;
```

```
113 }
114
115 void calculTier1(graphe* G) {
116
117
118     int i, j, k;
119     int p = 7500;
120
121     for(i = debTier1; i < finTier1 ; i++)
122     {
123         for(j = debTier1; j < finTier1; j++)
124         {
125             if(i != j && !verifSiSommetInListe(G, i, j))
126             {
127                 k = rand()%1000;
128                 if(k < p)
129                 {
130                     k = rand()%(poidsMaxTier2 - poidsMinTier2 + 1) + poidsMinTier2;
131
132                     G->list[i][j] = k;
133                     G->list[j][i] = k;
134                     G->I.compteur[i][0]++;
135                     G->I.compteur[j][0]++;
136                 }
137              }
138          }
139     }
140 }
141
142 void calculTier2(graphe* G){
143
144
145     int i, j;
146     int p, noeuds, k;
147
148     for(i = debTier2; i < finTier2; i++)
149     {
150         // pour les arc vers le tier precedent
151         p = rand()%2 + 1;
152         for(j = 0; j < p; j++)
153         {
154                 noeuds = calcul_noeuds(G, i, debTier1, finTier1, 100, tier2, tier1);
155                 k = rand()%(poidsMaxTier2 - poidsMinTier2 + 1) + poidsMinTier2;
156                 if(noeuds != -1)
157                 {
158                     G->list[i][noeuds] = k;
159                     G->list[noeuds][i] = k;
160                 }
161         }
162
163         // pour les arc vers le tier current
164         for(j = 0; G->I.compteur[i][tier2] < G->I.proba[i][0] && j <
        G->I.proba[i][0]; j++)
165         {
166             if(test_noeuds_max(G, i, noeudsMaxTier2, tier2))
167             {
168                 noeuds = calcul_noeuds(G, i, debTier2, finTier2, noeudsMaxTier2,
```

```
                tier2, tier2);
169             k = rand()%(poidsMaxTier2 - poidsMinTier2 + 1) + poidsMinTier2;
170             if(noeuds != -1)
171             {
172                 G->list[i][noeuds] = k;
173                 G->list[noeuds][i] = k;
174 
175                 G->I.compteur[i][1]++;
176                 G->I.compteur[noeuds][1]++;
177             }
178         }
179     }
180  }
181 }
182 
183 void calculTier3(graphe* G){
184 
185     int i, j, k;
186     int p, noeuds;
187 
188     for(i = debTier3; i < finTier3; i++)
189     {
190         // pour les arc vers le tier precedent
191         p = 2;
192         for(j = 0; j < p; j++)
193         {
194                 noeuds = calcul_noeuds(G, i, debTier2, finTier2, 100, tier3, tier2);
195                 k = rand()%(poidsMaxTier3 - poidsMinTier3 + 1) + poidsMinTier3;
196 
197                 if(noeuds != -1)
198                 {
199                     G->list[i][noeuds] = k;
200                     G->I.compteur[i][tier2]++;
201 
202                     G->list[noeuds][i] = k;
203                     G->I.compteur[noeuds][tier3]++;
204                 }
205         }
206 
207         // pour les arc vers le tier current
208         for(j = 0; G->I.compteur[i][tier3] < G->I.proba[i][0] && j <                ↵
            G->I.proba[i][0]; j++)
209         {
210             if(test_noeuds_max(G, i, noeudsMaxTier3, tier3))
211             {
212                 noeuds = calcul_noeuds(G, i, debTier3, finTier3, noeudsMaxTier3,    ↵
                    tier3, tier3);
213                 k = rand()%(poidsMaxTier3 - poidsMinTier3 + 1) + poidsMinTier3;
214 
215                 if(noeuds != -1)
216                 {
217                     G->list[i][noeuds] = k;
218                     G->I.compteur[i][tier3]++;
219 
220                     G->list[noeuds][i] = k;
221                     G->I.compteur[noeuds][tier3]++;
222                 }
```

```
223                 }
224             }
225         }
226     }
227
```

4.3 Vérification de la connexité

Fichiers pour vérifier la connexité du graphe :

```c
/**
 * Algorithme de recherche en profondeur
 * Les algorithmes implémentés permettent de vérifier la connexité du
 * graphe en appliquant l'algorithme de recherche en profondeur
 * */

/**
 * Parcours du graphe par chaque sommet
 * @param Graphe
 * @param entier
 * @param couleur
 * @param pere
 * */
void parcours_sommet(graphe* G, int s, int *couleur, int *pere);

/**
 * Parcours le graphe en profondeur pour savoir s'il est connexe
 * @param graphe
 * @return si connexe
 * */
int parcours_graphe(graphe* G);

```

```c
#include "const.h"
#include "graph.h"

#include <stdio.h>
#include <stdlib.h>

void parcours_sommet(graphe* G, int s, int *couleur, int *pere){
    int i;
    couleur[s] = 1;
    for(i = 0; i < TAILLE_GRAPHE ; i++)
        if(i != s && couleur[i] == 0 && G->list[s][i])
        {
            pere[i] = s;
            parcours_sommet(G, i, couleur, pere);
        }
    couleur[s] = 2;
}

int parcours_graphe(graphe* G){
    int i;
    int *couleur = calloc(TAILLE_GRAPHE,sizeof(int)); // 0 est blanc, 1 gris et 2 noir

    int *pere = malloc(sizeof(int)*TAILLE_GRAPHE);
    for(i = 0; i < TAILLE_GRAPHE ; i++)
        pere[i] = TAILLE_GRAPHE;

    int compteur = 0;
    for(i = 0; i < TAILLE_GRAPHE ; i++)
        if(couleur[i] == 0)
        {
            parcours_sommet(G, i, couleur, pere);
            compteur++;
        }

    free(couleur);
    free(pere);
    return compteur;
}

```

4.4 Création de la table de routage

Fichiers pour la création de la table de routage :

```c
#include "const.h"
/**
 * Algorithme de recherche du plus court chemin dans une graphe
 * */

/**
 * Structure de donnée pour sauvegarder la table de root (2 matrices)
 * */
typedef struct {
    int succ[TAILLE_GRAPHE][TAILLE_GRAPHE];
    int poids[TAILLE_GRAPHE][TAILLE_GRAPHE];
} routage;

/**
 * Initialisation de la table de rootage
 * @param graphe
 * @param taille du graphe
 * */
routage* init(graphe* G, int taille);

/**
 * Application de l'alogorthme Floyd Warshall
 * @param routage table de routage à remplir
 * @param taille du graphe
 * */
void Floyd_Warshall(routage* R, int taille);

/**
 * Libération de mémoire de la table de rootage
 * @param table de root
 * */
void libere_routage(routage* R);

/**
 * Affiche le plus court chemin entre une paire de noeuds
 * @param table de root
 * @param premier noeud de départ
 * @param dernier noeud d'arrivé
 * */
void afficher_chemin(routage* R, int deb, int fin);

```

```c
#include "const.h"
#include "graph.h"
#include "routage.h"

#include <stdlib.h> // rand ()

#include <stdio.h> // printf ()

#define inf 9999


routage* init(graphe* G, int taille) {
    int i, j;
    routage* R = malloc(sizeof(routage));


    for(i = 0; i < taille; ++i)
    {
        for(j = 0; j < taille; ++j)
        {
            R->poids[i][j] = G->list[i][j];

            if(R->poids[i][j] == -1)
            {
                R->poids[i][j] = inf;
                R->succ[i][j] = -1;
            }
            else
            {
                R->succ[i][j] = j;
            }
        }
    }

    for(i = 0; i < taille; ++i)
    {
        R->poids[i][i] = 0;
        R->succ[i][i] = i;
    }
    return R;
}

void Floyd_Warshall(routage* R, int taille) {
    int i, j, k;

    for(k = 0; k < taille; k++)
    {
        for(i = 0; i < taille; i++)
        {
            for(j = 0; j < taille; j++)
            {
                if(R->poids[i][k] != inf && R->poids[k][j] != inf
                    && (R->poids[i][j] > (R->poids[i][k] + R->poids[k][j])) )
                {
                    R->poids[i][j] = R->poids[i][k] + R->poids[k][j];
                    R->succ[i][j] = R->succ[i][k];
                }
```

```c

            }
        }
    }
}

void afficher_chemin(routage* R, int deb, int fin) {
    int stock_deb = deb;
    int voisin[TAILLE_GRAPHE] = {-1};

    int i, suiv = R->succ[deb][fin];
    for(i = 0; suiv != fin && i < TAILLE_GRAPHE; i++)
    {
        deb = R->succ[deb][fin];
        voisin[i] = deb;
        suiv = R->succ[deb][fin];
    }

    if(suiv == fin)
    {
        voisin[i] = fin;

        printf("\nChemin de %d à %d :\n%d", stock_deb, fin, stock_deb);
        for(int j = 0 ; j < i+1; j++)
        {
            printf(" -> %d", voisin[j]);
        }
        printf("\n");
    }
    else
    {
        printf("error\n");
    }
}

void libere_routage(routage* R) { free(R); }

```

4.5 Affichage

Fichiers pour l'affichage graphe :

```c
/**
 * Gestion de l'Affichage sous forme de fenetre graphique
 * */
#include "moteur_graphique.h"

/**
 * Affichage des connexions entre les noeuds
 * @param graphe
 * @param objet
 * @param cercle
 * @param les noeuds
 * @param le début du tier
 * @param la fin du tier
 * */
void afficher_voisin(graphe* G, flame_obj_t * fo, cercle_t * c,int sommet, int deb, int fin);

/**
 * Sauvegarde des données dans le tableau cercle pour l'affichage
 * Chaque tiers ont des couleurs différentes (tier 1 : rouge, tier2 :
 * bleu, tier 3 : vert)
 * @param cercle
 * @param le début du tier
 * @param la fin du tier
 * @param x réprésente les coordonnées en abscisse
 * @param y représente les coordonnées en ordonnée
 * */
void init_affichage_tier(cercle_t * c, int debut,int fin, int tier,int * x,int * y);

/**
 * Permet l'initialisation des objets graphiques:
 * initialisation des cercles et des connexions des noeuds*
 * @param graphe
 * @param objet
 * @param cercle
 * */
void initialisation_objets_graphique(graphe *G,flame_obj_t * fo,cercle_t * c);

/**
 * Recherche l'identifiant du cercle dans la fenetre graphique
 * @param x abscisse
 * @param y ordonnée
 * @return identifiant du cercle
 * */
int trouve_id(int x,int y);

/**
 * Affiche le plus court chemin
 * @param objet
 * @param la table de routage
 * @param cercle
 * @param l'identifiant de l'expediteur
 * @param l'identifiant du destinataire
 * @param la couleur
 * */
void affiche_chemin(flame_obj_t * fo,routage* R, cercle_t * c,int deb, int fin,enum couleur coul);
```

```
56
57  /**
58   * Permet la gestion de l'interaction utilisateur et machine
59   * @param graphe
60   * @param objet
61   * @param la table de routage
62   * @param cercle
63   * */
64  void interaction_user(graphe * G,flame_obj_t * fo,routage * R,cercle_t * c);
65
66  /**
67   * Fonction principale
68   * Gestion de la fenetre graphique
69   * @param graphe
70   * @param la table de routage
71   * */
72  void gestion_fenetre_graphique(graphe* G, routage *R);
73
```

```c
#include <stdio.h>

//~ local
#include "const.h"
#include "graph.h"
#include "moteur_graphique.h"
#include "routage.h"

void afficher_voisin(graphe* G, flame_obj_t * fo, cercle_t * c,int sommet, int deb, int fin)
{
    int j;
    for(j = deb; j < fin; j++) if(G->list[sommet][j] != -1) { afficher_connexion(fo,c,sommet,j,GRIS); }
}


void init_affichage_tier(cercle_t * c, int debut,int fin, int tier,int * x,int * y)
{
    int i;

    for( i = debut ; i < fin ; i ++)
    {
        c[i].rad = TAILLE_CERCLE;
        if(tier == tier1)
        {
            colorer_cercle(&c[i],ROUGE);
            c[i].pos_x = *x * ( 3 * TAILLE_CERCLE);
            c[i].pos_y = *y * ( 3 * TAILLE_CERCLE);

        }
        if(tier == tier2)
        {
            colorer_cercle(&c[i],BLEU);
            c[i].pos_x = *x * ( 3 * TAILLE_CERCLE);
            c[i].pos_y = *y * ( 3 * TAILLE_CERCLE);
        }
        if(tier == tier3)
        {
            colorer_cercle(&c[i],VERT);
            c[i].pos_x = *x * ( 3 * TAILLE_CERCLE);
            c[i].pos_y = *y * ( 3 * TAILLE_CERCLE);
        }
        *x+=1;
        if(*x > 10)
        {
            *y += 1;
            *x = 1;
        }
    }
}

void initialisation_objets_graphique(graphe *G,flame_obj_t * fo,cercle_t * c)
{
    int noeud;

    int x = 1;
```

```
56      int y = 1;
57
58      // > Initialise les coordonnées des cercles
59      init_affichage_tier(c,debTier1,finTier1,tier1,&x,&y);
60      init_affichage_tier(c,debTier2,finTier2,tier2,&x,&y);
61      init_affichage_tier(c,debTier3,finTier3,tier3,&x,&y);
62
63      for (noeud = 0; noeud < TAILLE_GRAPHE; noeud++)
64      {
65          afficher_cercle(fo, &c[noeud]);
66          // > Affichage des connexions mais c'est illisble
67          //~ afficher_voisin(G, fo, c, noeud, debTier1, finTier1);
68          //~ afficher_voisin(G, fo, c, noeud, debTier2, finTier2);
69          //~ afficher_voisin(G, fo, c, noeud, debTier3, finTier3);
70      }
71  }
72
73  int trouve_id(int x,int y)
74  {
75      x -= (2 * TAILLE_CERCLE);
76      y -= (2 * TAILLE_CERCLE);
77      return ((y / (3 * TAILLE_CERCLE))*10) + (x / (3*TAILLE_CERCLE));
78  }
79
80  void affiche_chemin (flame_obj_t * fo,routage* R, cercle_t * c,int deb, int
    fin,enum couleur coul)
81  {
82      int voisin[TAILLE_GRAPHE] = {-1};
83
84      int i = 0, suiv = R->succ[deb][fin];
85      voisin[i] = deb;
86
87      for(i = 1; suiv != fin && i < TAILLE_GRAPHE; i++)
88      {
89          deb = R->succ[deb][fin];
90          voisin[i] = deb;
91          suiv = R->succ[deb][fin];
92      }
93
94      if(suiv == fin)
95      {
96          voisin[i] = fin;
97
98          int d, e, j;
99          for(j = 0 ; j < i; j++)
100         {
101             d = voisin[j]; e = voisin[j+1];
102             afficher_connexion(fo, c, d, e, coul);
103         }
104     }
105     else
106     {
107         printf("error\n");
108     }
109 }
110
111
```

```
112  void interaction_user(graphe * G,flame_obj_t * fo,routage * R,cercle_t * c)
113  {
114      XEvent event;
115      int cmp = 0;
116      int id_1 = 0;
117      int id_2 = 0;
118      int save_id_1 = 0;
119      int save_id_2 = 0;
120
121      int click_x, click_y;
122
123      while (1)
124      {
125          if (XPending(fo->display) > 0)
126          {
127            XNextEvent(fo->display, &event);
128            if(recupere_clavier(event) == 'q') { break; }
129            if (event.type == ButtonPress)
130            {
131                  // Récupere les coordonées
132                  click_x = event.xkey.x;
133                  click_y = event.xkey.y;
134
135                  // colorie et affiche les cercles
136                  colorer_cercle(&c[trouve_id(click_x,click_y)],JAUNE);
137                  afficher_cercle(fo,&c[trouve_id(click_x,click_y)]);
138
139                  if(cmp == 0)
140                  {
141                      save_id_1 = id_1;
142                      save_id_2 = id_2;
143                      id_1 = trouve_id(click_x,click_y);
144
145                      // Permet d'éffacer les traits
146                      affiche_chemin ( fo, R, c, save_id_1, save_id_2, NOIR);
147                      affiche_croix(fo, c[save_id_2].pos_x, c[save_id_2].pos_y, NOIR);
148                      initialisation_objets_graphique ( G, fo, c);
149
150                      // Coloris les cercles
151                      colorer_cercle( &c[id_1], JAUNE);
152
153                      // Affiche les cercles
154                      afficher_cercle(fo, &c[id_1]);
155                  }
156                  else
157                  {
158                      id_2 = trouve_id(click_x,click_y);
159
160                      // Coloris les cercles
161                      colorer_cercle( &c[id_2], JAUNE);
162
163                      // Affiche les cercles
164                      afficher_cercle(fo, &c[id_2]);
165                  }
166                  cmp ++;
167            }
168        }
```

```
169         else
170         {
171           if(cmp == 2)
172           {
173               cmp = 0;
174
175               // Affiche les chemins
176               afficher_chemin (R, id_1, id_2);
177               affiche_chemin (fo, R, c, id_1, id_2, JAUNE);
178
179               // Affiche debut
180
181               // Affiche arrivé
182               affiche_croix(fo, c[id_2].pos_x, c[id_2].pos_y, BLANC);
183           }
184         }
185     }
186   }
187
188   void gestion_fenetre_graphique(graphe* G, routage *R)
189   {
190       // > Initialisation du canvas:
191       flame_obj_t * fo = init_canvas();
192
193       // > Allocation de mémoire de la structure de donnée d'un cercle
194       cercle_t c[TAILLE_GRAPHE];
195
196       // > Initialise les connexions et les cercles
197       initialisation_objets_graphique(G,fo,c);
198
199       // > Gestion des interaction utilisateur machine
200       interaction_user(G,fo,R,c);
201
202       // > Fermeture du canvas
203       flame_close(fo);
204   }
205
```

```c
#include "flame.h"

#define TAILLE_ECRAN_HAUTEUR 500
#define TAILLE_ECRAN_LARGEUR 500

#define TAILLE_CERCLE 15

/* Définir la couleur 8 bits */
typedef unsigned char byte;

/* Structure de données d'un cercle */

typedef struct cercle_s {

  double pos_x;
  double pos_y;

  double rad;

  //Couleur du cercle
  byte r;
  byte g;
  byte b;

} cercle_t;

/* Enumération des couleurs */
enum couleur{
    ROUGE,
    VERT,
    BLEU,
    BLANC,
    JAUNE,
    GRIS,
    NOIR
};

/* Permet d'afficher un cercle rempli*/
void remplir_cercle(flame_obj_t *fo, cercle_t *b);

/* Permet d'afficher un cercle */
void afficher_cercle(flame_obj_t *fo, cercle_t *b);

/* Permet d'afficher les connexions */
void afficher_ligne(flame_obj_t *fo,int x1,int y1,int x2,int y2);

/* Permet de récuperer les entrées du clavier */
char recupere_clavier(XEvent event);

/* Colorier un cercle */
void colorer_cercle(cercle_t * c,enum couleur coul);

/* Initialise la fenetre graphique */
flame_obj_t * init_canvas();

/* Permet d'afficher les connexion entre les objets symbolisé par un trait */
void afficher_connexion(flame_obj_t *fo,cercle_t * c,int id_1,int id_2,enum
```

```
    couleur coul);
58
59  /* Permet l'affichage d'une croix */
60  void affiche_croix(flame_obj_t *fo,  int x, int y, enum couleur coul);
61
```

```c
/**
 * Bibliothèque de gestion d'un graphe:
 * Cette bibliothèque permet de dessiner des cercles, des lignes et des croix.
 * Elle a été créer afin de manipuler des graphes simplements.
 * */

#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>
#include <inttypes.h>

#include "flame.h"
#include "moteur_graphique.h"

#include "pi.h"

#define A_STEP 0.01


void remplir_cercle(flame_obj_t *fo, cercle_t *b)
{
    flame_set_color(fo, b->r, b->g,b->b);
    for (double angle = 0.0; angle < 2 * PI; angle += A_STEP)
    {
        flame_draw_line(fo, b->pos_x, b->pos_y, b->pos_x + b->rad * cos(angle),
        b->pos_y + b->rad * sin(angle));
    }
}

void afficher_cercle(flame_obj_t *fo, cercle_t *b)
{
    flame_set_color(fo, b->r, b->g,b->b);
    for (double angle = 0.0; angle < 2 * PI; angle += A_STEP)
    {
        flame_draw_point(fo, b->pos_x + b->rad * cos(angle), b->pos_y + b->rad *
        sin(angle));
    }
}

void afficher_ligne(flame_obj_t *fo,int x1,int y1,int x2,int y2)
{
    flame_draw_line(fo, x1, y1,x2,y2);
}

char recupere_clavier(XEvent event)
{
    char c;
    if (event.type == KeyPress)
    {
        c = XLookupKeysym(&event.xkey, 0);
        return c;
    }
    return -1;
```

```
 56  }
 57
 58  flame_obj_t * init_canvas()
 59  {
 60      return flame_open("Graphe",TAILLE_ECRAN_HAUTEUR,TAILLE_ECRAN_LARGEUR);
 61  }
 62
 63  void colorer_cercle(cercle_t * c,enum couleur coul)
 64  {
 65      switch(coul)
 66      {
 67          case ROUGE :
 68          {
 69              c->r = 255;
 70              c->g = 0;
 71              c->b = 0;
 72              break;
 73          }
 74          case BLEU :
 75          {
 76              c->r = 0;
 77              c->g = 0;
 78              c->b = 255;
 79              break;
 80          }
 81          case VERT :
 82          {
 83              c->r = 0;
 84              c->g = 255;
 85              c->b = 0;
 86              break;
 87          }
 88          case BLANC :
 89          {
 90              c->r = 255;
 91              c->g = 255;
 92              c->b = 255;
 93              break;
 94          }
 95          case JAUNE :
 96          {
 97              c->r = 255;
 98              c->g = 255;
 99              c->b = 0;
100              break;
101          }
102          default:
103          {
104              perror("switch");
105              exit(EXIT_FAILURE);
106          }
107      }
108  }
109
110  void afficher_connexion(flame_obj_t *fo, cercle_t * c, int id_1, int id_2, enum
     couleur coul)
111  {
```

```
112       if(coul == BLANC) flame_set_color(fo, 255, 255, 255);
113       if(coul == ROUGE) flame_set_color(fo, 255, 0, 0);
114       if(coul == GRIS) flame_set_color(fo, 200, 200, 200);
115       if(coul == NOIR) flame_set_color(fo, 0, 0, 0);
116       if(coul == JAUNE) flame_set_color(fo, 255, 255, 0);
117       afficher_ligne(fo, c[id_1].pos_x, c[id_1].pos_y, c[id_2].pos_x, c[id_2].pos_y);
118   }
119
120   void affiche_croix(flame_obj_t *fo, int x, int y, enum couleur coul) {
121       if(coul == BLANC) flame_set_color(fo, 255, 255, 255);
122       if(coul == NOIR) flame_set_color(fo, 0, 0, 0);
123
124       afficher_ligne(fo, x - 10, y - 10, x + 10, y + 10);
125       afficher_ligne(fo, x - 10, y + 10, x + 10, y - 10);
126   }
127
```

4.6 Les autres fichiers

4.6.1 Constantes

Fichier pour les définitions des constantes :

```c
/**
 * Ensemble de définition des constantes
 * */

#define TAILLE_GRAPHE 100
#define tier1 0
#define tier2 1
#define tier3 2

#define nbTier1 8
#define nbTier2 20
#define nbTier3 72
#define ZERO 0

#define debTier1 0
#define finTier1 8
#define debTier2 8
#define finTier2 28
#define debTier3 28
#define finTier3 100

#define poidsMaxTier1 10
#define poidsMinTier1 5
#define poidsMaxTier2 20
#define poidsMinTier2 10
#define poidsMaxTier3 50
#define poidsMinTier3 15

#define noeudsMaxTier2 3
#define noeudsMaxTier3 1

```

4.6.2 Bibliothèque flame11

Code source de la bibliothèque flame11 crée par Yaspr, voici le lien github :

`https://github.com/yaspr/flame11`

```c
#ifndef FLAME_H
#define FLAME_H

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>

//
typedef struct flame_obj_s {

  GC        gc;
  Window    window;
  Display  *display;
  Colormap  colormap;
  int       fast_color_mode;

} flame_obj_t;

int  flame_close(flame_obj_t *fo);
int  flame_event_waiting(flame_obj_t *fo);
void flame_flush(flame_obj_t *fo);
void flame_clear_display(flame_obj_t *fo);
void flame_draw_point(flame_obj_t *fo, int x, int y);
flame_obj_t *flame_open(char *title, int width, int height);
char flame_wait(flame_obj_t *fo, int *click_x, int *click_y);
void flame_set_color(flame_obj_t *fo, int red, int green, int blue );
void flame_draw_line(flame_obj_t *fo, int x1, int y1, int x2, int y2);
void flame_clear_color(flame_obj_t *fo,  int red, int green, int blue );

#endif

```

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>

#include "flame.h"

//
int flame_flush_display(flame_obj_t *fo)
{
  if (fo && fo->display)
    return XFlush(fo->display), 1;
  else
    return 0;
}

//
int flame_close(flame_obj_t *fo)
{
  if (fo)
  {
    XFreeGC(fo->display, fo->gc);
    XCloseDisplay(fo->display);

    free(fo);

    return 1;
  }
  else
    return 0;
}

//
flame_obj_t *flame_open(char *title, int width, int height)
{
  XEvent e;
  int vb = 0;
  Visual *visual;
  int blackColor, whiteColor;
  XSetWindowAttributes attrib;
  flame_obj_t *fo = malloc(sizeof(flame_obj_t));

  fo->display = XOpenDisplay(0);

  if (!fo->display)
    {
      fprintf(stderr,"flame_open: unable to open the graphics window.\n");
      exit(1);
    }

  visual = DefaultVisual(fo->display, 0);

  fo->fast_color_mode = (visual && visual->class == TrueColor);

  blackColor = BlackPixel(fo->display, DefaultScreen(fo->display));
  whiteColor = WhitePixel(fo->display, DefaultScreen(fo->display));

```

```
58    //fo->window = XCreateSimpleWindow(fo->display, DefaultRootWindow(fo->display),
      0, 0, width, height, 0, blackColor, blackColor);
59    fo->window = XCreateSimpleWindow(fo->display, DefaultRootWindow(fo->display), 0,
      0, width, height, 0, blackColor, blackColor);
60
61    attrib.backing_store = Always;
62
63    XChangeWindowAttributes(fo->display, fo->window, CWBackingStore, &attrib);
64
65    XStoreName(fo->display, fo->window,title);
66
67    XSelectInput(fo->display, fo->window, StructureNotifyMask | ExposureMask |
      KeyPressMask | ButtonPressMask | ButtonReleaseMask | PointerMotionMask);
68
69    XMapWindow(fo->display, fo->window);
70
71    fo->gc = XCreateGC(fo->display, fo->window, 0, 0);
72
73    fo->colormap = DefaultColormap(fo->display, 0);
74
75    XSetForeground(fo->display, fo->gc, whiteColor);
76
77    while(!vb)
78      {
79        XNextEvent(fo->display, &e);
80
81        vb = (e.type == MapNotify);
82      }
83
84    return fo;
85  }
86
87  //
88  void flame_draw_point(flame_obj_t *fo, int x, int y)
89  { XDrawPoint(fo->display, fo->window, fo->gc, x, y); }
90
91  //
92  void flame_draw_line(flame_obj_t *fo, int x1, int y1, int x2, int y2)
93  { XDrawLine(fo->display,fo->window,fo->gc, x1, y1, x2, y2); }
94
95  //
96  void flame_set_color(flame_obj_t *fo, int r, int g, int b)
97  {
98    XColor color;
99
100   if (fo->fast_color_mode)
101     color.pixel = ((b & 0xff) | ((g & 0xff) << 8) | ((r & 0xff) << 16));
102   else
103     {
104       color.pixel = 0;
105       color.red   = r << 8;
106       color.green = g << 8;
107       color.blue  = b << 8;
108       XAllocColor(fo->display, fo->colormap, &color);
109     }
110
111   XSetForeground(fo->display, fo->gc, color.pixel);
```

```
112  }
113
114
115  //
116  void flame_clear_display(flame_obj_t *fo)
117  { XClearWindow(fo->display, fo->window); }
118
119  //
120  void flame_clear_color(flame_obj_t *fo, int r, int g, int b )
121  {
122    XColor color;
123    XSetWindowAttributes attrib;
124
125    color.pixel = 0;
126    color.red   = r << 8;
127    color.green = g << 8;
128    color.blue  = b << 8;
129    XAllocColor(fo->display, fo->colormap, &color);
130
131    attrib.background_pixel = color.pixel;
132    XChangeWindowAttributes(fo->display, fo->window, CWBackPixel,&attrib);
133  }
134
135  //
136  int flame_event_waiting(flame_obj_t *fo)
137  {
138    XEvent event;
139
140    flame_flush_display(fo);
141
142    while (1)
143      {
144        if (XCheckMaskEvent(fo->display, -1, &event))
145      {
146        if (event.type == KeyPress)
147          {
148            XPutBackEvent(fo->display, &event);
149
150            return 1;
151          }
152        else
153          if (event.type == ButtonPress)
154            {
155          XPutBackEvent(fo->display, &event);
156
157          return 1;
158            }
159          else
160            return 0;
161      }
162        else
163      return 0;
164      }
165  }
166
167  //
168  char flame_wait(flame_obj_t *fo, int *click_x, int *click_y)
```

```
169  {
170    XEvent event;
171
172    flame_flush_display(fo);
173
174    while (1)
175      {
176        if (XPending(fo->display) > 0)
177      {
178        XNextEvent(fo->display, &event);
179
180        if (event.type == KeyPress)
181          {
182            /* printf("%c\n", XLookupKeysym(&event.xkey, 0)); */
183            return XLookupKeysym(&event.xkey, 0);
184          }
185        else
186          if (event.type == ButtonPress)
187            {
188          *click_x = event.xkey.x;
189          *click_y = event.xkey.y;
190
191          //Left click == 1, Right click == 3
192          return event.xbutton.button;
193            }
194      }
195        else
196      {
197        ;
198      }
199      }
200  }
201
202
```

```
1  #ifndef PI_H
2  #define PI_H
3
4  #define PI 3.14159265359
5  #define PI_2 PI / 2.0
6  #define PI_3 PI / 3.0
7
8  #endif
9
```