PROJET - Programmation répartie

Partie 1. Application pour la mise en place d'un graphe à colorier

1. Mise en place du problème

Pour répondre au problème, on doit d'abord relier les noeuds entre-eux. Chaque sommet est relié à ses voisins dans le graphe (représenté par une arête).

On a alors besoin d'avoir un serveur qui gère les liaisons entre les sommet selon le graphe que lui seul connait. Pour cela on considère que chaque client (noeud) sait qui il est, et envoie alors son indice et son adresse au serveur. Celui-ci va alors transmettre à chaque site les coordonnées des sites avec lesquels il doit communiquer.

Cela permet donc de relier les noeuds entre eux en respectant le graphe donné. Ainsi, une fois que le serveur à "distribué" les arêtes, nous n'en avons plus besoin.

Pour assurer le fait qu'une connexion ne se fasse qu'une fois (et pas deux fois, avec une dans les deux sens), pour une arête x-y, nous envoyons à x l'ordre de se connecter à y, et a y d'attendre.

Pour la réalisation de ce projet, notre dossier .zip contient les fichiers suivants : noeuds.c, serveur.c, parseur.c ; ainsi que structures.c et fonctions.c. Ces deux dernier fichiers contienent respectivement les structures et les fonctions separées pour une meilleure lisibilité du code.
Un readme est disponible à la fin du rapport en page 5.

2. Parseur

Pour commencer, nous avons crée un parseur qui va venir lire le fichier contenant le graphe. Ce parseur va récupérer le nombre de sommets et d'arêtes du graphe, puis va venir lire chaque arête du graphe et les insérer dans un tableau pour, pouvoir ensuite répartir les arêtes aux processus concernés. On a choisi d'utiliser un tableau de structure pour la liste des arêtes. La structure est de la forme :

```
struct aretes{
    int noeud1;          //noeud numero 1 de l'arete avant ->
    int noeud2;          //noeud numero 2 de l'arete apres ->
};
```

Elle contient les deux noeuds constituants chaque arêtes.

3. Rôle du serveur

Le serveur a pour rôle de mettre en place les communications entre les différents sites, pour respecter le graphe donné. Pour cela, il va recevoir toutes les informations des sommets que ces derniers fourniront dans une structure définie ci-après :

```
struct infos_graphe {
    int numero;                    //numero du noeud courant sur le graphe
    int descripteur;               //descripteur du noeud
    struct sockaddr_in adrProc;    //adresse du noeud courant
};
```

Ainsi, pour faire une communication entre deux sites, il faut connaitre l'adresse du client (adresse IP + port) alors le serveur doit connaitre de quel noeud vient les informations. On ajoute alors le numero du sommet du graphe dans la structure. Le serveur avec ces deux informations va alors retransmettre les informations aux clients voisins.

Dans le fichier serveur.c, nous pouvons voir toutes les étapes que le serveur doit effectuer pour mettre en place les communications :

A CONSTRUCTION DU SERVEUR

ETAPE 1 *Récupération des informations du graphe* : Le serveur va utiliser le parseur pour récupérer toutes les informations qui serviront à la construction du graphe ; le nombre de sommets, le nombre d'arêtes, ainsi que toutes les arêtes.

ETAPE 2 *Récupérations des informations sur le voisinage* : Une fois que nous avons toutes les arêtes, nous construisons un second tableau où nous allons stocker tous les voisins de chaque noeud. Ce tableau nous servira plus tard lorsqu'il faudra envoyer à chaque sommet les informations pour contacter ses voisins.

ETAPE 3 *Gestion de la socket serveur* : On va créer une socket serveur qui permettra de discuter avec tous les différents client. C'est grâce à cette socket que le processus I enverra ses informations. Une fois celà fait, on va maintenant nommer la socket serveur. Le nommage est obligatoire car ici nous en avons besoin pour que nous puissions envoyer l'adresse des différents voisins à tous les noeuds.

ETAPE 4 *Mise en écoute de la socket serveur* : On met en écoute la socket pour que tous les noeuds (socket) puisse s'y connecter.

B RECEPTION DES INFORMATIONS DES SOMMETS

ETAPE 5 *Acceptation du noeud* : Le serveur accepte la demande de connexion d'un noeud pour qu'il puisse recevoir ses informations.

ETAPE 6 *Reception des informations du client* : Pour chaque noeud, on va construire une structure qui va accueillir les informations reçues par le noeud courant. Cette structure va être stockée dans le tableau des informations crée par le serveur. Ainsi, la case à l'indice i contient la structure contenant les informations du noeud i+1 (car les numéros des noeuds commencent à 1 et non 0).

C ENVOI DES INFORMATIONS AUX SOMMETS

ETAPE 7 *Envoi du nombre de voisins aux sommets* : Juste après avoir reçu les information de connexion, le serveur envoie au sommet connecté son nombre de voisin. chaque noeud va alors créer autant de socket que de voisin.

ETAPE 8 *Envoi des informations de connexion* : Le serveur va maintenant coordonner la connexion entre les arêtes pour éviter les interblocages. Pour chaque arête x-y du graphe, le serveur va envoyer à x les informations de y, pour que x puisse envoyer une demande de connexion à y.

D EXTINCTION DU SERVEUR

ETAPE 9 *Fermeture de la socket serveur* : Après avoir donné toutes les informations aux sommets, ces derniers sont alors autonomes pour la suite des opérations. Nous n'avons donc plus besoin du serveur ; on ferme sa socket et le programme s'arrête.

5. **Rôle des sommets**

Pour cette partie, il va nous falloir lancer autant de fois le programme noeud.c qu'il y a d'arêtes. Nous avons écrit un script python qui va lancer autant de noeuds que contient le graphe passé en paramètre. Ainsi, on pourra voir à la suite les données de chaque noeuds sur un même terminal. Pour une question de lisibilité, chaque affichage est accompagné du numéro du noeud ainsi que d'une couleur fixe pour pouvoir différencier les affichages de chauqe noeuds plus facilement dans le terminal.

Une fois la mise en place de faite, regardons le rôle des sommets :

A **CONSTRUCTION D'UN SOMMET**

ETAPE 1 *Création de la socket du noeud* : Cette socket permettra de discuter avec le serveur. C'est grace à cette socket que le noeud enverra ses informations. On va ensuite nommer la socket pour qu'on puisse envoyer l'adresse du sommet au serveur.

ETAPE 2 *Désignation de la socket serveur* : Cette désignation nous permet de pouvoir se connecter au serveur. On peut faire cette désignation grâce à l'adresse IP et au port du serveur qu'on a donné en paramètre.

ETAPE 3 *Demande de connexion* : Cette étape est assez explicite, le sommet i fait une demande de connexion au serveur et attends que le serveur accepte la connexion.

B **CONSTRUCTION DE LA SOCKET DE RECEPTION DES VOISINS**

ETAPE 4 *Gestion de la socket* : Ici on va créer une socket qui va être destinée à recevoir les informations des noeuds voisins. Elle va donc être destiné à être le serveur de ses voisins qui demande une connexion. On va ensuite nommer cette socket, nommage qui habituellement n'est pas obligatoire mais ici nous en avons besoin pour pouvoir recevoir les informations des noeuds voisins.

ETAPE 5 *Mise en écoute de la socket* : Pour être sur que la socket soit prête lorsque nous aurons besoin de l'utiliser, nous la mettons en écoute en amont des demandes de connexion.

C **ENVOI LES INFORMATIONS AU SERVEUR**

ETAPE 6 *Envoyer les informations au serveur* : On commence d'abord par rassembler les informations dont nous avons besoin dans la structure énoncée plus haut. Ainsi on récupère l'indice du sommet en paramètre ainsi que l'adresse de la socket après le nommage. On envoie alors toutes les informations au serveur.

D **RECEPTION DES INFORMATIONS DU SERVEUR**

ETAPE 7 *Reception du nombre de voisins* : On construit une structure qui va accueillir les informations reçues par le serveur. Dans un premier temps, chaque sommet reçoit son nombre de voisin.

ETAPE 8 *Réception des sommets à contacter* : En même temps que le nombre de voisin, le sommet a également reçu le nombre de sommet auquel il doit se connecter. Typiquement, si il y a une arête x-y, on va demander à x de se connecter à y. Chaque sommet x va ensuite recevoir sa liste (s'il en possède une) de sommet y à contacter pour pouvoir former les xy-arêtes du graphe. Après cette étape, nous n'avons plus besoin de la socket serveur, nous la fermons.

E **MISE EN CONTACT DES SOMMETS**

ETAPE 9 *Création des structures pour les connexions* : Ici, on va créer toutes les variables dont on a besoin pour connecter les sommets entre-eux ; le tableau des thread, ainsi que les structures pour envoyer (respectivement recevoir) les informations des voisins sortant (respectivement entrants).

ETAPE 10 *Demandes de connexion* : Les sommets ayant reçu des ordre de connexion vont alors créer une socket pour pouvoir envoyer les informations à leur voisins. Ils vont utiliser la socket du noeud reçue par le serveur pour s'y connecter et former une arête en informant le noeud voisin de leur voisinage.

ETAPE 11 *Création de thread pour communiquer avec les voisins sortants* : Une fois que le voisin a accepté la demande, il faut lui envoyer un message pour lui informer que l'on est son voisin. On créer pour cela un thread dans lequel on va garder la connexion et envoyer un message au voisin sortant. Il pourra dès à présent communiquer avec nous grâce à la socket actuelle.

ETAPE 12 *Réception des demandes* : Dans une seconde phase, chaque sommet va alors, à l'aide de sa socket mise précédemment en attente, recevoir une, plusieurs ou aucune demande de connexion. On va accepter ces demandes venant de notre voisin entrant et établir le lien entre chaque sommet reliés par des arêtes.

ETAPE 13 *Création du thread pour communiquer avec les voisins entrants* : Pour finir, on va recevoir les informations du voisin qui vient de nous contacter. On créer un thread pour pouvoir recevoir le message du voisin entrant, et garder la connexion jusqu'a que l'on ait besoin de lui envoyer un message, ou vice-versa.

On utilise le protocole TCP avec une phase de connexion/acceptation, puis une phase d'envoi/réception de message.

6. Rôle des threads utilisés

Pour faire du multi-processus, nous avons utilisé deux types de thread avec deux fonctionnements différents qu'on a mis dans une seule fonction qui traite les deux cas :

THREAD - **DEMANDEUR** : Ce thread va être appelé lorsqu'un noeud x d'une arête x-y veut entrer en contact avec le noeud y. la connexion ayant déjà été établie au préalable, ce thread va dans un premier temps servir à envoyer les informations du noeud courant au voisin que l'on contacte (numéro du noeud). Dans la suite du projet, ce thread restera ouvert pour pouvoir à tout moment envoyer ou recevoir des informations venant des voisins.

THREAD - **ACCEPTEUR** : Le noeud va appeler ce thread lorsqu'il vient de recevoir des connexions, l'idée est pour après de recevoir le numero du voisin entrant et son adresse. De même que le thread ci-dessus, ce thread restera ouvert pour pouvoir à tout moment envoyer ou recevoir des informations aux voisins.

2. Résumé

Suite à ces étapes, nous venons de mettre en place un réseau de processus/noeuds interconnectés ayant la forme d'un graphe connexe. Dans la prochaine partie nous utiliserons cette interconnexion pour résoudre le problème de la k-coloration dans un graphe.

Partie 2 : Algorithme pour le problème de coloration de sommets

Sommaire

2.1 Rappel du problème de coloration de graphe 5
2.2 Modification de la partie 1 5
2.3 Ajout de l'algorithme a notre code 6
2.4 Utilisation des threads/multiplexage 7
2.5 Résultats de l'algorithme 8
2.5.1 Calcul du nombre de messages envoyés 8
2.5.2 Ordre de grandeur 8
2.5.3 Test sur des grands graphes 8
2.6 Problèmes rencontrés 9
2.6.1 Fermeture de socket non voulue 9
2.6.2 Voisins d'ordre inférieur non colorés 9
2.6.3 Erreur dans la taille de message 9
2.7 Utilisation de support et gestion de projet 10

3 README : Guide de lancement du programme 11

2.1 Rappel du problème de coloration de graphe

Depuis les premières coloration de graphes planaires pour représenter des cartes, la coloration de sommet à toujours été un sujet central des mathématiques. La coloration de sommets au sein d'un graphe consiste à colorer tout les sommets en une couleur, de telle sorte à ce que les sommets reliés par une arête soient de couleur différente. Pour un graphe G, le nombre chromatique $\chi(G)$ correspond alors au plus petit nombre de couleur qu'il est possible d'utiliser pour colorer le graphe.

Après avoir mis en place notre réseau distribué pour reproduire la structure d'un graphe donné, nous devons à présent faire en sorte que chaque noeud puisse indépendamment déterminer une coloration minimale du graphe en utilisant l'envoi de messages à ses voisins. Nous ajoutons alors dans le fichier *noeuds.c*, une seconde partie qui sera entièrement dédiée à la coloration du graphe. La première partie servant uniquement à connecter les processus entre-eux, nous avons cependant dû la modifier.

2.2 Modification de la partie 1

Pour commencer, nous avons finallement opté pour du multiplexage pour que les noeuds puissent se connecter et en s'accepter mutuellement. Nous avons donc pour chaque noeud un tableau de voisin à qui ils doivent demander une connexion. Ainsi, cette première étape se fait en 3 parties : L'interaction avec le serveur pour récupérer le nombre de voisin; La connexion d'un noeud à un autre; L'acceptation d'un noeud par un autre.

Lors de l'interaction avec le serveur, nous avons ajouté un attribut dans la structure *nbVois* :

```
struct nbVois{
    int nbNoeuds;         //nb de noeuds total dans le graphe
    int nbVoisinTotal;    //nb de voisin au total d'un noeud
    int nbVoisinDemande; //nb de voisin pour la connexion
};
```

Cet ajout, nous permettra lors de la seconde partie de s'arrêter quand le graphe sera totalement coloré, mais nous y reviendrons plus tard. On reçoit également du serveur lors de cette premiere partie, un ordre que nous expliquerons plus bas. Ainsi, qu'un signal de la part du serveur nous permettant d'attendre que les connexions entre les noeuds soient totalement effectuées avant de commencer la partie 2.

Lors de la connexion d'un noeud, à part enlever le thread correspondant à un autre, on a rien changé concernant la structure de code. On a ajouté un envoie des informations du noeud courant (numero, ordre, adresse,...) au noeud avec qui je viens de me connecter, pour pouvoir par la suite le rajouter dans le tableau des informations des voisins du noeud courant.

Ainsi, lors de l'acceptation d'un nœud par un autre, on ajoute dans le tableau des informations des voisins du noeud courant *info_voisin* de type infos_graphe, les informations du noeud qu'on vient de recevoir apres l'acceptation du noeud en question (ajout de l'ordre du noeud par rapport a la partie 1) :

```
struct infos_graphe {
    int numero;                    //numero du noeud courant
    int ordre;                     //ordre du noeud courant
    int descripteur;               //descripteur du noeud
    struct sockaddr_in adrProc;    //adresse du noeud courant
};
```

A la fin de cette première partie, on a donc un tableau des informations utiles regroupées dans la structure suivante : de tous les voisins de chaque noeud. De plus, ils ont chacun un ordre de coloration pour la suite.

2.3 Ajout de l'algorithme a notre code

Nous avons expliqué les modifications de la partie 1 et pouvons à présent travailler à la suite de cette dernière. Nous proposons alors d'implémenter l'algorithme suivant : (pour la suite, on suppose que $ordre_i$ est notre ordre de coloration (l'ordre 1 se colore en premier, puis le 2, etc), nous reviendrons dessus plus loin) :

Variable pour chaque noeud :
$tableauCouleurVoisin_i[nbVoisin_i]$ initialisé à 0;
$couleurMax_i = 0$;
$dernierOrdreFini = 0$;

Lorsque je recois un message :
$couleurMax_i = MAX(couleurMax_i, couleur_j)$;
Si le message est de type <COULEUR, $ordre_j$, $couleur_j$ >
    J'ajoute $couleur_j$ à $tableauCouleursVoisins_i$;

si $ordre_j > dernierOrdreFini$
    $\forall k \in N_G(i)$: envoyer <INFO, $ordre_j$, $couleur_j$ >;

Si $dernierOrdreFini + 1 = ordre_i$
    Je me colorie;

Fonction coloration :
$couleur_i = 1$;
$voisin = 0$;
tant que $voisin < |N_G(i)|$
    Si $tableauCouleursVoisins_i[voisin] = couleur_i$
        $couleur_i$++;
        $voisin = 0$;
    Sinon
        $voisin$++;
$\forall k \in N_G(i)$: envoyer <COULEUR, $ordre_i$, $couleur_i$ >;

Dans le pseudo code ci-dessus, nous constatons que nous avons deux types de messages : COULEUR, et INFO. Le premier est envoyé par un sommet (qui vient de se colorer) à l'ensemble de ses voisins ; le second est propagé dans l'ensemble du graphe pour que le sommet d'ordre $ordre_j + 1$ sache que c'est à son tour de se colorer.

Les messages envoyés correspondent à une structure *message* avec trois membres : *requete* pour le type du message, *ordreI* pour l'ordre du noeud, et *message* pour envoyer la couleur. Dans cette seconde partie, seul ce type de message sera envoyé.

Par ailleurs, le nombre de couleurs utilisées pour colorer le graphe évolue tout au long de l'algorithme, dès qu'une couleur plus grande que la $couleurMax_i$ est utilisée et donc envoyée aux autres noeuds, cette dernière est mise à jour. Comme l'information de la coloration se propage dans tout le graphe, cela garantit que chaque noeud i ait bien la même valeur pour $couleurMax_i$ que ce soit au cours du processus ou a la fin, quand on a la coloration.

Pour l'ordre des sommets $ordre_i$ que nous avons évoqué plus haut, celui ci est déterminé grace au degré de chaque sommet : plus un sommet a de voisins, plus ce dernier sera prioritaire dans la coloration et aura donc un ordre le plus petit possible. Ainsi, le serveur va se servir du tableau d'arêtes récupéré par le parseur à la partie 1 pour pouvoir calculer le degré de tous les noeuds du graphe. Ce calcul sera effectué dans un tableau qui sera ensuite trié par degré décroissant.

Le serveur va ensuite attribuer l'ordre de chaque sommet en fonction de son nombre de voisins dans le tableau; plus ce dernier est grand, plus l'ordre du sommet sera prioritaire. En cas d'égalité, le numero du noeud est utilisé pour les départager. Ce système permet de garantir le même ordre à chaque exécution de notre programme sur un même graphe. L'ordre ne pourra donc pas être aléatoire, et il est démontré que cet ordre de coloration est algorithmiquement plus efficace qu'un ordre aléatoire.

2.4 Utilisation des threads/multiplexage

Nous avons utilisé un thread pour appliquer l'algorithme de Coloration énoncé plus haut. Nous allons expliquer un peu plus en détail les structures de données passées en paramètre ainsi qu'éventuellement des détails sur le thread *Coloration*.

Premièrement, nous voyons dans l'algorithme de Coloration que nous avons besoin de l'ordre du noeud courant, ainsi que des informations de ses voisins, pour pouvoir leur envoyer la couleur du nœud courant et du numéro du nœud pour un eventuel affichage. C'est pourquoi nous mettons tout cela en parametre du thread dans la structure :

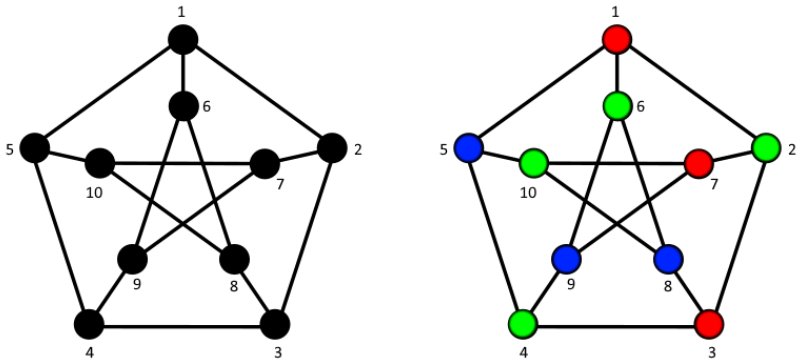
```
struct paramsColoration {
    int numero;                                 //numero du noeud courant
    int ordre;                                  //ordre du noeud
    int nbVoisins;                              //nombre de voisins du noeud
    struct couleurVoisin* couleurVoisins;       //tableau couleurs voisins
    struct infos_Graphe* VoisinsCourant;        //informations du voisins
};
```

Comme nous l'avons expliqué plus haut, ce thread permet d'exécuter en parallèle la réception des messages des différents noeuds, et l'algorithme de coloration. Pour ce dernier, chaque noeud commence à la couleur 1, on parcourt alors le tableau des couleurs de voisin pour savoir si un de nos voisin possède la même couleur. Si c'est le cas, on "augmente" la couleur ; si aucun voisin ne posssède cette couleur, on se colore, puis on l'envoie à tout nos voisins un part un.

Par ailleurs, pour la réception de message, comme chaque noeud doit pouvoir en réceptionner à tout moment, nous avons donc utilisé du multiplexage pour le traitement des différents messages. Lorsqu'on reçoit un message de type COULEUR, le noeud met à jour son tableau des couleurs de ses voisins. Sinon, le type de message est INFO et le noeud renvoie la couleur en mettant à jour sa couleur maximum, et également la variable $dernierOrdreFini_i$ s'il n'était pas déjà à jour. On vérifie ensuite si c'est au noeud de se colorer ou non.

Comme *dernierOrdreFini* est l'ordre du plus grand noeud duquel nous savons qui s'est coloré. Cela nous garantit, qu'une fois que le dernier noeud nous aura fait savoir qu'il s'est coloré et que le message aura circulé dans tout le graphe, chaque sommet va pouvoir afficher la valeur de $couleurMax$ pour avoir la k-coloration calculée sur le graphe choisi.

2.5 Résultats de l'algorithme



(a) Graphe de Petersen (b) Graphe coloré

Figure 1: Graphe de Petersen coloré par l'algorithme de coloration distribué

En prenant des graphes quelconques comme K_5 (la clique de taille 5) ou encore le graphe de Petersen, nous pouvons voir que notre algorithme renvoie bien la coloration minimale pour ces graphes (respectivement 5 et 3), apres avoir executé nos codes. Par exemple, en déroulant l'algorithme sur le graphe de Petersen (a) sur la figure 1 a la page 8, nous avons une coloration minimale qui est $\chi(P) = 3$ (b).

2.5.1 Calcul du nombre de messages envoyés

En prenant des instances du site, le programme prend un peu plus de temps pour trouver une coloration. Nous pouvons calculer le nombre d'envoi maximum de message pour une instance de graphe donnée :

Soit un graphe G, et $\Delta(G)$ le degré maximum de G (le plus grand nombre de voisin parmi les sommets) que nous abregerons Δ. Sachant que chaque noeud doit se colorer, toutes les actions qui vont suivre sont donc effectuées n-fois.

Pour chaque noeud, il y aura dans un premier temps, Δ messages de type <COULEUR, $ordre_j$, $couleur_j$> qui seront envoyés. Dans un second temps, chaque noeud du graphe va recevoir un message de type COULEUR pour l'avertir de la coloration du sommet d'ordre $ordre_j$, et va ensuite renvoyer cette information à tous ses voisins avec le type INFO, ce qui fais $(n-1)\Delta$ messages par npoeud.
Comme nous avons vu que ce processus doit se répéter pour les n noeuds, nous pouvons borner le nombre d'envoi de message par $O(n(\Delta + (n-1)\Delta)) = O(\Delta n^2)$.

2.5.2 Ordre de grandeur

Pour avoir un ordre de grandeur, le fichier $DSJC500.1$ à un $\Delta = 68$, ce qui fait $68 \times 500^2 = 17$ millions de messages. En pratique, il est plus juste d'estimer ce nombre à l'aide de $\overline{deg}(G)$, le degré moyen des sommets de G. Si l'on reprend le fichier $DSJC500.1$, nous avons alors $50 \times 500^2 = 12{,}5$ millions de messages, soit $1,36$ fois moins.

2.5.3 Test sur des grands graphes

Pour l'instance $DSJC500.1$, on trouve une 21-coloration du graphe en moins de deux minutes ($\tilde{2}0$ secondes pour colorer 100 noeuds). En reprenant le calcul précédent, on peut alors estimer la vitesse d'envoi des messages à $\frac{12{,}5\times10^6}{100}$, soit un peu plus de 125 mille messages par seconde. Cette coloration n'est pas optimale, mais l'utilisation d'un algorithme NP-Complet pour trouver $\chi(G)$ prendrait un temps exponentiel comparé à notre algorithme.

Avec en entrée le graphe $DSJC250.5$, notre programme met environ **50 secondes** pour trouver une coloration avec notre algorithme, puis trouve une 42-coloration du graphe.

2.6 Problèmes rencontrés

Nous avons au cours de ce projet rencontré plusieurs problèmes que nous avons essayé de résoudre au fur et à mesure de notre progression.

2.6.1 Fermeture de socket non voulue

Pour commencer, lors de la première partie nous avions au début utilisé les threads pour que les nœuds se connectent et s'acceptent entre eux, et pour l'envoi de message entre les nœuds également.

Nous n'avons pas abouti cette idée car nous avions des noeuds qui sortaient de leurs programmes. Cela fermait leurs sockets, car elles avaient fini de faire leurs travails, avant que d'autres noeuds ne puissent avoir fini les leurs, et donc nous avions des fermetures de socket assez souvent. C'est ce qui nous a conduit à faire du multiplexage à la place.

2.6.2 Voisins d'ordre inférieur non colorés

Un second problème que nous avons rencontré concerne l'ordre d'arrivé des messages. En effet, il arrivait qu'un noeud reçoive un message de type INFO avant le même message de type COULEUR. Cela se produisait dans le cas où, le noeud i a un voisin d'ordre $ordre_i - 1$. Ce voisin va envoyer le message de type COULEUR à tout ses voisins.

Or, si l'un de ses voisins était très rapide, et envoyait le même message de type INFO directement à notre noeud i, juste avant qu'il recoive le message de type COULEUR, alors le noeud i partait se colorer (car il est le suivant selon l'ordre). Mais comme il n'avais pas reçu le message de type COULEUR de son voisin, son tableau de couleur des voisins n'était pas à jour, et la coloration n'était donc pas bonne.

C'est pourquoi lorsque c'est à un noeud de se colorer, il faut que le noeud vérifie que tout ses voisins plus petits que lui (tourjours en terme d'ordre), soient différents de 0 (signifiant que le voisin n'est pas coloré). Si cette condition est respectée, on peut alors se colorer. Sinon, on attend le message de type COULEUR de notre voisin. A la fin, cela assurera qu'aucun noeud ne se colore sans avoir reçu les couleurs de tout ses voisins avant lui.

2.6.3 Erreur dans la taille de message

Un autre problème, est celui qui consiste à avoir des messages attendus qui ne sont pas de la bonne taille par rapport à ce que la fonction recv reçoit. Ainsi, nous avons pu constater que certains messages avaient une taille trop petite par rapport à celle attendue. Cependant comme nous n'avons qu'un seul type de message dans la seconde partie du programme, les messages devraient tous avoir la même taille.

Ce problème entrainait une fermeture prématurée de la socket d'un noeud, et par réaction en chaine la chute de tout les processus formant le graphe distribué.
Nous en avons déduit que le problème venait sûrement des paramètres des threads, mais lorsque nous affichons les valeurs pour essayer de déboguer le code, ceux ci paraissaient cohérents avec les données qu'on passait en parametres. Nous avions des messages qui arrivaient avec par exemple, un $type_j = 12$, alors que nous n'envoyions que des messages de type 0 ou 1 (défini par alias avec COULEUR et INFO). Nous avons déduit que cette erreur devait être due à une *corruption* du message lors des send. Ainsi, nous avons essayé de regler le probleme en filtrant le type de message qui arrivait pour ne pas traiter les messages qui n'ont pas le bon type.

Indéniablement, cela a créé un problème de manque de communication, du fait qu'il manque des messages, étant donné que nous ne les traitons pas quand ils sont d'un type incoherents. Or si les messages que l'on ne renvoie pas ou que l'on ne traite pas sont de type COULEUR, alors nous aurons un problème ; car cela veut dire que certains voisins n'auront pas l'information que le noeud courant est coloré.

Cependant, nous avons montré précédemment qu'un noeud peut se colorer seulement si tous ses voisins d'ordres plus petit que lui se sont colorés. Ce problème entraine donc une sorte d'interbloquage, car un noeud va attendre un message du noeud ayant l'ordre juste avant lui, et comme ce message à été corrompu, il n'arrivera donc jamais.

C'est pourquoi actuellement, nous ne sommes pas en mesure de garantir que le programme termine et donne une coloration du graphe donnée dans tous les cas. La corruption devant être due au trop grand nombre de messages envoyés, la multiplication du nombre de message entraine une augmentation de la probabilité qu'une corruption intervienne sur le réseau. Nous avons néanmoins réussi à aller au bout de certaines exécutions du programme, et c'est comme cela que nous avons réussi à fournir des valeurs de coloration pour les fichiers avec 250 et 500 noeuds.

2.7 Utilisation de support et gestion de projet

Pour ce projet, nous avons mis en place un Git pour pouvoir collaborer et apporter des modifications au projet tout en gardant un suivi détaillé sur sa progression. Depuis le début de projet en octobre, nous avons veillé à faire au minimum une réunion par semaine pour mettre en commun nos idées et avancer ensemble sur le projet, et nous avons également veillé à utiliser la même nomenclature concernant les variables, fonctions, etc.

3 README : Guide de lancement du programme

Un readme est présent dans le même dossier que ce rapport et contient les instructions nécessaires au lancement du programme. Voici une version plus détaillée.
Pour lancer la création du graphe, il faut commencer par lancer le fichier **serveur.py**, avec en arguments le *fichier du graphe* sur lequel on va créer notre réseau, pour pouvoir lire les arêtes nécessaires à la mise en place du graphe, et le *port du serveur*. Le script python se charge également de compiler les fichiers *.c* et de créer les dossiers nécessaires au lancement des scripts.

Pour lancer les différents noeuds, il faut lancer le script **noeuds.py**, avec en arguments l'ip du serveur, le port du serveur, ainsi que le graphe sur lequel on va créer notre réseau (pour créer le nombre de procéssus/noeuds adéquat). Ce script va lancer le nombre de noeud que possède le graphe, à l'aide de la commande :

```
$./bin/noeud id_serveur port_serveur port_noeud numero_noeud
```

Le dernier paramètre *numero_noeud* n'intervient pas dans la coloration du graphe et n'a pour but que d'aider à la visualisation et à la comprehension lors des divers affichages.

Utilisation :

```
$python3 serveur.py fichier_graphe port_serveur

$python3 noeuds.py fichier_graphe ip_serveur port_serveur
```

Exemple d'utilisation avec notre dossier rendu :

```
#lancement du serveur dans le terminal 1
$python3 serveur.py graphes/graphe_petersen.txt 5000

#lancement des noeuds dans le terminal 2
$python3 noeuds.py graphes/graphe_petersen.txt 127.0.0.1 5000
```