



HAI721I - Programmation répartie

Projet 2^{ème} partie : Rapport

Département informatique

Jérémie BENTOLILA 21906253

Maxime BOURRET 21901688

28 décembre 2022

Table des matières

1	Introduction	1
1.1	Présentation du sujet	1
1.2	Rappel : La coloration de graphe	1
1.3	Exécution	2
2	Conception	3
2.1	Modifications sur la première partie	3
2.2	Réflexion sur la coloration distribuée	5
3	Implémentation	6
3.1	Coloration	6
3.2	Difficultés rencontrées	7
4	Conclusion	8
4.1	Résultats	8

1 Introduction

1.1 Présentation du sujet

Ce projet est réalisé dans le cadre du module HAI721I et a pour objectif de résoudre le problème de la coloration de sommets. Celui-ci est organisé en deux parties. Nous avons traité la première portant sur la création du graphe, ce rapport traite la seconde qui porte sur la coloration du graphe. Le code source du projet est disponible sur [GitHub](#).

L'objectif de cette partie est de développer un algorithme de coloration distribué suite à la mise en place de ce graphe effectuée en première partie. Pour rappel, ces graphes se traduisent par un réseau de processus interconnectés en fonction d'une description donnée en entrée. Cette description est définie sous le format des instances fournies à l'adresse : <http://cedric.cnam.fr/~porumbed/graphs/>.

L'application doit donc permettre de produire n'importe quelle instance suivant ce format. Chaque instance est représentée de la manière suivante : le premier caractère (c, p, ou e) de la ligne représente le contenu de cette dernière comme suit :

```
c <commentaire>
p edge <nombre_noeuds> <nombre_aretes>
e <noeud1> <noeud2>
```

En effet, "c" désigne les lignes de commentaire, "p" se trouve une seule fois dans chaque fichier et sert à décrire le nombre de nœuds et d'arêtes que contient le graphe. Quant à "e", il désigne une arête entre 2 nœuds.

1.2 Rappel : La coloration de graphe

La coloration de graphe est un problème NP-difficile (problème pour lequel on peut vérifier une solution en temps polynomial, mais pour lequel il n'existe pas d'algorithme permettant de trouver une solution optimale en temps polynomial). Celui-ci consiste à attribuer des couleurs à chaque sommet d'un graphe de manière à ce que les sommets adjacents (c'est-à-dire reliés par une arête) ne partagent pas la même. Le nombre minimal de couleurs nécessaires pour colorier un graphe de manière satisfaisante est appelé sa chromatique.

Par exemple, considérons les graphes suivants :

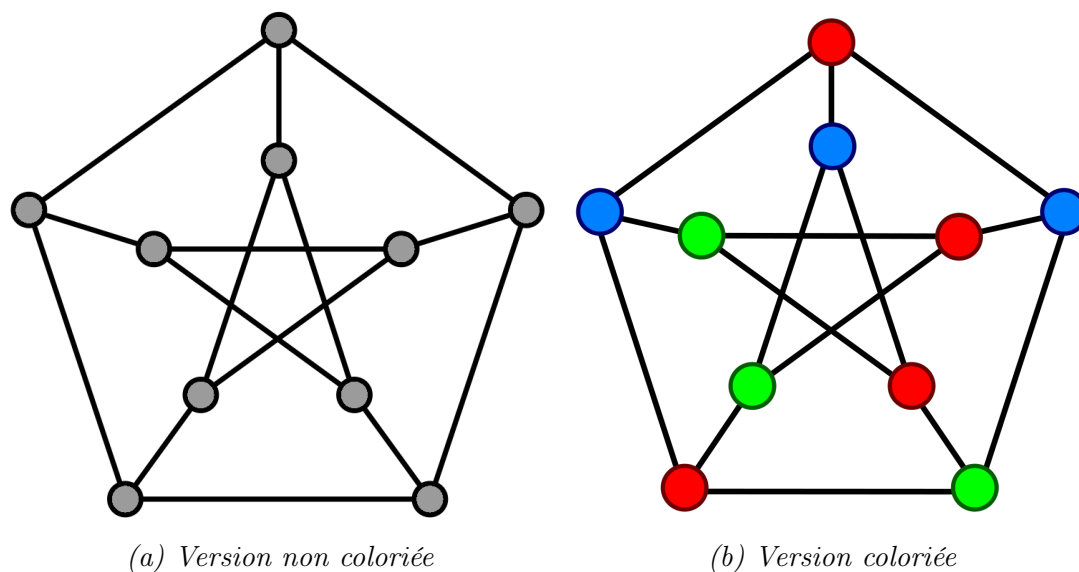


FIGURE 1 – Graphe de Petersen

Nous voyons ici que pour colorier ce graphe de manière satisfaisante, nous devons utiliser au moins trois couleurs, par exemple le rouge, le bleu et le vert. Ici le nombre de couleurs utilisé est minimal.

La coloration de graphe est un problème important en informatique car elle peut être utilisée pour résoudre de nombreux autres problèmes, notamment la planification de vols pour éviter les conflits dans l'espace aérien, l'attribution des registres en compilation et bien d'autres.

1.3 Exécution

Afin de télécharger et de lancer notre projet, ouvrez un terminal puis exécutez les commandes suivantes :

```
git clone https://github.com/MqxBrT/HAI721I-Projet
cd HAI721I-Projet
make run
```

Un `Makefile` est inclus dans les fichiers téléchargés et permet donc de compiler et de lancer le projet en une commande. Vous devez ensuite choisir la vitesse d'exécution et fournir un nom de fichier contenu dans le dossier *files*. Ce dernier doit contenir un graphe décrit avec la structure évoquée dans la partie 1.1 ci-dessus. Le programme va ensuite s'exécuter en lançant le serveur sur le port 10 000 par défaut et en démarrant les nœuds. Si vous souhaitez simplement le compiler, exécutez la commande `make compile`.

2 Conception

Après avoir effectué la présentation orale de la première partie du projet portant sur l'interconnexion des sommets, des modifications nous ont été suggérées avant de commencer la deuxième partie. Nous avons donc commencé par corriger notre implémentation afin de partir sur des bonnes bases pour la coloration. Par la suite, nous avons choisi, contrairement à la première partie, de réfléchir sur papier avant de commencer à coder. Cela nous a permis de repérer la plupart des problèmes de notre algorithme et de les corriger sans incidence sur le code ainsi que de mieux aborder la partie implémentation car le déroulement était plus clair. Nous avons ensuite développé l'algorithme de coloration en C. Grâce à ce que nous avons appris lors de la première partie, nous avons eu moins de problèmes de compréhension lors de l'implémentation. Enfin, une fois le projet fini et testé à de nombreuses reprises, nous avons rédigé le rapport.

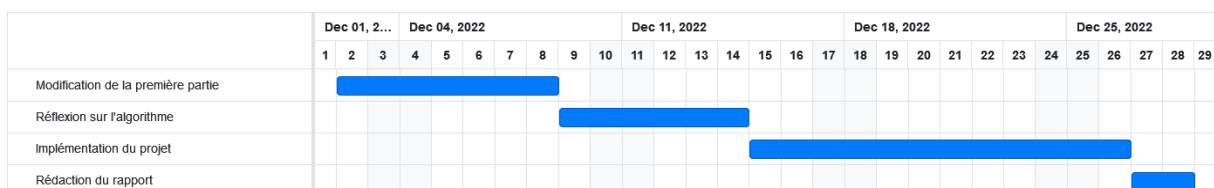


FIGURE 2 – Diagramme de Gantt du déroulement de la deuxième partie du projet

2.1 Modifications sur la première partie

Nous avons commencé la deuxième partie de projet en corrigeant les points soulevés lors de notre présentation orale. Tout d'abord les processus communiquent maintenant en simple connexion via une seule et unique socket. Premièrement les échanges entre les nœuds ont été modifiés. Le parseur différencie maintenant les voisins entrants et sortants. Les degrés correspondants sont transmis aux sommets et ces derniers créent les sockets d'écoute et de connexion en fonction. Deuxièmement avec le serveur, le second contact pour envoyer les adresses se fait avec la même connexion que lors du premier contact de découverte des processus nœuds. Les communications suivantes (confirmation d'interconnexion et résultat de la coloration) utilisent aussi cette socket. Nous avons aussi modifié les échanges pour que toutes les informations envoyées par le serveur tiennent dans une structure et ainsi envoyer un seul message au nœud au lieu de trois entiers dans des messages individuels.

```
struct sInformations {
    int entrants; // nombre de voisins entrants (à écouter)
    int sortants; // nombre de voisins sortants (à se connecter)
    int flag; // 1 si on a le token au début de la coloration, 0 sinon
};
```

Nous avons également modifié l'implémentation lors des `bind()` afin que la socket soit associée à l'IP et au port publics permettant ainsi d'exécuter les processus sur des machines différentes tout en gardant la possibilité de se connecter avec `127.0.0.1`. Selon l'architecture sur laquelle le programme est lancé il est possible de rencontrer l'erreur suivante :

```
eno1: erreur lors de la recherche d'infos sur l'interface:  
Périphérique non trouvé
```

En effet, afin de récupérer l'IP nous utilisons une suite de commandes impliquant `ifconfig`. Or la balise contenant l'IP peut changer (sur les machines de l'Université il s'agit de `'eno1'`, sur x2Go il s'agit de `'ens160'`). Pour corriger le problème, il vous faut donc remplacer les occurrences de `'eno1'` dans les fichiers `serveur.c` et `commun.h` par la balise qui vous est affichée lors de l'exécution de `ifconfig`.

La dernière modification porte sur le serveur. En effet ce dernier ne se ferme plus une fois les adresses envoyées, il attend un message des nœuds qui lui confirment leur interconnexion et par la suite il reçoit toutes les couleurs choisies. Ces modifications n'influent en rien la coloration et ont pour seul but un affichage plus clair des différentes étapes traversées par les nœuds.

2.2 Réflexion sur la coloration distribuée

Afin de réaliser la seconde partie du projet, nous avons d'abord commencé les réflexions sur papier et avons mis au point un pseudo algorithme afin de nous guider lors de l'implémentation. Pour commencer nous avons décidé que le serveur donnerait un token au sommet avec le plus haut degré. Ce token donne le droit d'effectuer la coloration et va circuler parmi tous les nœuds. Nous avons différencié les différents cas de figures possibles que nous avons regroupé en deux catégories. Les opérations internes qui concernent la coloration du sommet et le passage du token. Les opérations externes qui dépendent des messages reçus avec la gestion des suivants et des voisins colorés.

```
Si j'ai le token :  
    Demande couleurs  
Sinon :  
    Demande token  
Tant que je n'ai pas rendu le token ou que je ne suis pas coloré :  
    Pour tous les voisins entrants :  
        Tant qu'un message est reçu :  
            Si le voisin vient de se colorer :  
                Enregistrer le voisin  
            Si le voisin demande ma couleur :  
                Donner ma couleur  
            Si le voisin demande le token :  
                Ajouter le voisin aux suivants  
            Si je reçois sa couleur :  
                Enregistrer sa couleur  
            Si je reçois le token :  
                Enregistrer le token  
                Si je ne suis pas coloré :  
                    Enregistrer le prédécesseur  
                    Demander couleurs  
    Pour tous les voisins sortants :  
        [Similaire aux voisins entrants]  
    Si des voisins se sont colorés pendant cette itération :  
        Supprimer les voisins colorés des suivants  
    Si j'ai reçu toutes les couleurs de mes voisins et que je ne suis pas coloré :  
        Je me colore  
        Je dis à tous mes voisins que je suis coloré  
    Si je suis coloré et que j'ai le token :  
        Si j'ai un suivant :  
            Donner le token au suivant  
        Sinon si j'ai un prédécesseur :  
            Donner le token au prédécesseur  
Donner ma couleur au serveur
```

FIGURE 3 – Version simplifiée de l'algorithme de coloration en pseudo-code

3 Implémentation

Nous avons repris le script Python que nous avons utilisé pour la première partie et l'avons modifié pour ajouter l'option du choix de la vitesse de lancement des processus. Le programme serveur a un paramètre en plus lors du lancement qui vaut 0 si on veut qu'il n'effectue aucun sleep. Nous avons donc les deux commandes suivantes pour exécuter un serveur et un nœud.

```
./bin/serveur numero_port nom_fichier sleep_time  
./bin/noeud ip_serveur numero_port id_noeud
```

L'arborescence des fichiers avec *commun.h* et *parseur.h* reste inchangée ainsi que le code couleur des messages affichés lors de l'exécution :

- **Orange** : message affiché par l'exécution du parseur
- **Violet** : message affiché par l'exécution du serveur
- **Bleu** : message affiché par l'exécution d'un nœud
- **Vert** : message de succès affiché par un nœud lorsqu'il est connecté à tous ses voisins ou lorsqu'il affiche sa couleur
- **Rouge** : message d'erreur affiché avant l'arrêt d'un processus

3.1 Coloration

Une fois le plan établi et l'algorithme sur papier assez clair, nous avons commencé l'implémentation. La plupart du travail a été fait dans le fichier *noeud.c*, seules quelques communications ont été rajoutées au fichier *serveur.c*.

Une fois interconnecté avec ses voisins, chaque nœud va commencer par demander la couleur de tous ses voisins s'il dispose du token et va demander le token sinon. Il va ensuite rentrer dans la boucle `while()` principale de laquelle il sortira uniquement lorsqu'il sera coloré ainsi que tous ses voisins. Dans cette dernière, les opérations externes sont traitées en premier. Il y a d'abord un parcours itératif des voisins entrants. Si un message est reçu une action est effectuée selon le contenu de ce dernier. Nous avons choisi de communiquer en échangeant des structures ayant la forme suivante :

```
struct sMessage {  
    int demandeCouleur; // 1 si on veut la couleur, 0 sinon  
    int reponseCouleur; // 1 si on donne notre couleur, 0 sinon  
    int couleur;        // 0 si pas coloré, coloré sinon  
    int demandeToken;   // 1 si on veut le token, 0 sinon  
    int passageToken;   // 1 si on donne le token, 0 sinon  
    int estColore;      // 1 si on a fini sa coloration, 0 sinon  
};
```


Pour chaque message reçu on vérifie donc quelle balise est à 1, ce qui va permettre au nœud d'agir en fonction. On traite ensuite itérativement les voisins sortants de la même manière que sont traités des entrants.

Pour faire circuler le token parmi les sommets, chaque nœud a un tableau de suivant et un prédécesseur. Lorsqu'un voisin nous envoie un message de demande de token, on place ce dernier dans notre tableau de suivants. Afin de stocker les nœuds dans le tableau, nous avons mis en place la structure suivante :

```
struct sAdresse {  
    int tableau;    // 0 si entrants, 1 si sortants  
    int indice;    // indice dans le tableau  
};
```

Ainsi, on peut retrouver facilement la socket connectée au voisin correspondant. S'il s'agit d'un voisin entrant, l'attribut `tableau` vaut 0, sinon il vaut 1 et son indice dans le tableau est sauvegardé. Lorsqu'on reçoit le token, on stocke le prédécesseur avec cette structure de la même manière. Une fois toutes les couleurs voisines récupérées, la première couleur n'étant pas utilisée est choisie et les voisins sont prévenus. Le nœud donne alors le token à tous les éléments de sa liste de suivants pour enfin le rendre à son prédécesseur et terminer.

3.2 Difficultés rencontrées

Parmi les difficultés auxquelles nous avons fait face, il y a la réception de message sur une socket avec `recv()`. Cette fonction étant bloquante, il nous a fallu trouver un moyen de vérifier si un message était reçu, sans pour autant s'arrêter sur un buffer vide. Pour pallier ce problème nous avons utilisé la fonction `ioctl(sockets[i], FIONREAD, &status)` qui associe à `status` une valeur positive si un message est reçu sur `socket[i]` sans effectuer de blocage.

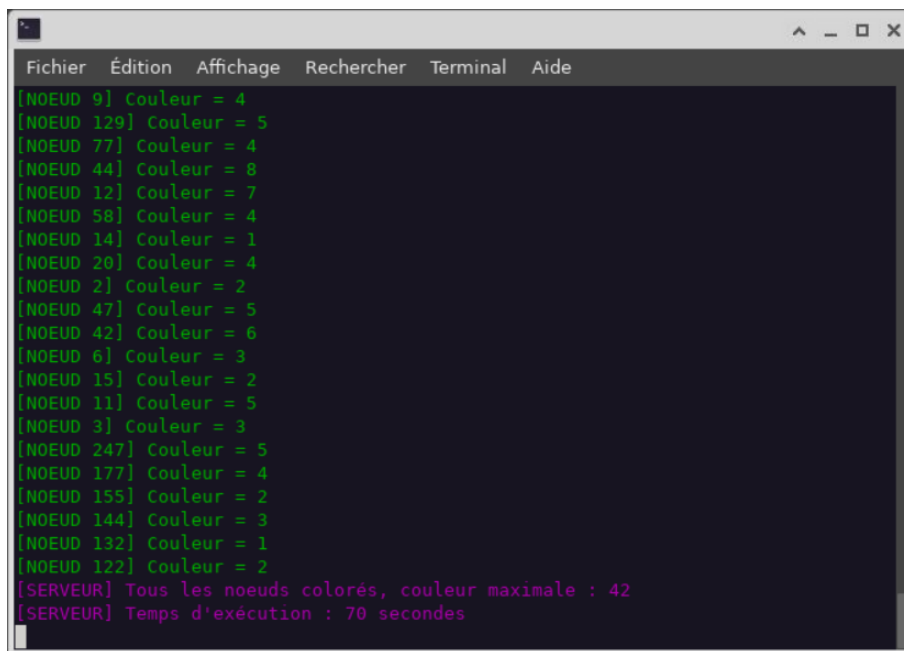
Le principal problème rencontré est la fin de vie du processus nœud. En effet, nous avons dû modifier l'implémentation plusieurs fois car certains nœuds quittaient la boucle `while()` alors qu'ils étaient encore dans liste de suivants d'un autre sommet. Nous avons alors essayé de ne demander le token qu'à un voisin, mais cela a soulevé un nouveau problème. Si le sommet qui obtenait le token du serveur n'avait aucune demande, la coloration devenait impossible car il ne savait à qui le donner. Nous avons donc fait en sorte que lorsqu'un sommet se colore, il broadcast cette information à tous ses voisins qui le retirent alors de leurs liste de suivants s'il en fait partie. Le nœud ne peut alors sortir de la boucle `while()` que s'il est coloré et que tous ses voisins le sont aussi et qu'il a rendu le token à son prédécesseur.

4 Conclusion

Dans cette seconde partie, nous avons poussé le développement réseau dans le langage C à des limites que nous n'avions jamais exploré avant. L'expérience accumulée sur la première partie nous a été utile afin de mieux cerner le problème avant la phase d'implémentation. Les connaissances accumulées nous ont permis d'aller plus vite sur des tâches pratiques telles que l'envoi de message ou la manipulation des structures. Les principales sources de problèmes ont été plus algorithmiques que techniques. Les réflexes de nommage clair des variables et fonctions ainsi que les commentaires réguliers de la première partie nous ont aidé à ne pas nous perdre dans le code et ont facilité le débogage et le travail en collaboration.

4.1 Résultats

En ce qui concerne les performances de notre programme, nous avons obtenu des résultats satisfaisants. L'algorithme que nous avons choisi n'est pas optimal mais la vitesse d'exécution est correcte même avec un nombre élevé de sommets (à noter que nous avons effectué nos tests sur des sessions via *x2Go*, la vitesse observée sur une machine locale est plus élevée). Pour le fichier `dsjc250.5`, notre programme trouve une solution en 70 secondes environ. Les résultats varient en fonction de l'ordre d'arrivée des sommets dans les tableaux de suivants mais avoisinent tous 40. Nous avons également effectué des tests sur le fichier `dsjc500.5` pour lequel nous avons obtenu des résultats autour de 70 en 260 secondes ainsi que sur le fichier `dsjc1000.1` où nous avons obtenu des solutions environnant 30 en 900 secondes en moyenne.



```
[NOEUD 9] Couleur = 4
[NOEUD 129] Couleur = 5
[NOEUD 77] Couleur = 4
[NOEUD 44] Couleur = 8
[NOEUD 12] Couleur = 7
[NOEUD 58] Couleur = 4
[NOEUD 14] Couleur = 1
[NOEUD 20] Couleur = 4
[NOEUD 2] Couleur = 2
[NOEUD 47] Couleur = 5
[NOEUD 42] Couleur = 6
[NOEUD 6] Couleur = 3
[NOEUD 15] Couleur = 2
[NOEUD 11] Couleur = 5
[NOEUD 3] Couleur = 3
[NOEUD 247] Couleur = 5
[NOEUD 177] Couleur = 4
[NOEUD 155] Couleur = 2
[NOEUD 144] Couleur = 3
[NOEUD 132] Couleur = 1
[NOEUD 122] Couleur = 2
[SERVEUR] Tous les noeuds colorés, couleur maximale : 42
[SERVEUR] Temps d'exécution : 70 secondes
```

FIGURE 4 – Résultat affiché dans le terminal après l'exécution sur le fichier `dsjc250.5`