PROJET TUTORÉ DE STRUCTURES DE DONNÉES
-
GRAPHES ET COMBINATOIRES

PROBLÈME DU FLOT MAXIMUM

IS2A3 - Lundi 31 mai 2021

Encadrante : Clarisse DHAENENS

Auteurs : Badmavasan KIROUCHENASSAMY
Caroline SCHMID

Table des matières

Chapitre 1

Contexte du projet

Pour résoudre un problème de flot maximum, plusieurs algorithmes peuvent être mis en œuvre. Nous étudions ici la mise en place de l'algorithme de **DINIC** ainsi que les structures de données nécessaires et l'arborescence des fichiers.

Le problème de flot maximum consiste à trouver le flot le plus élevé que l'on peut faire passer dans un réseau en respectant les capacités de chaque arc, soit le flot maximum que l'on peut faire passer par chaque arc.

L'algorithme de **DINIC** permet, à partir d'un flot initial, de rechercher une chaîne améliorante dans un réseau de façon à construire un flot de valeur supérieure. En réitérant cette opération un certain nombre de fois, le flot que l'on obtient devient maximum. L'algrithme prend en paramètre un graphe d'écart, il faut donc transformer le réseau du problème de flot maximum en un graphe d'écart pour trouver le flot maximum et revenir à un réseau répondant au problème posé.

On considère que le flot initial est de 0 et que les sommets sont numérotés de 1 à n, n étant le nombre de sommets.

Chapitre 2

Analyse

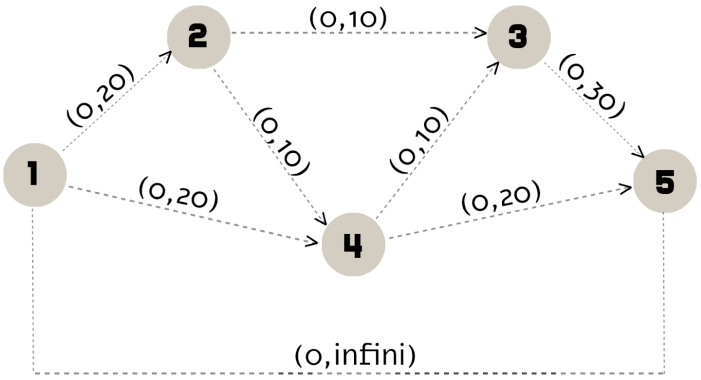
2.1 Représentation des données

La chaîne améliorante de l'algorithme de **DINIC** est la chaîne du plus court chemin en nombre d'arcs et sera donc trouvée par l'algorithme de parcours en largeur. Étant donné que ce dernier algorithme génère l'arborescence donnant le plus court chemin en nombre d'arc, on l'utilise pour trouver la chaîne améliorante de l'algorithme de **DINIC**.

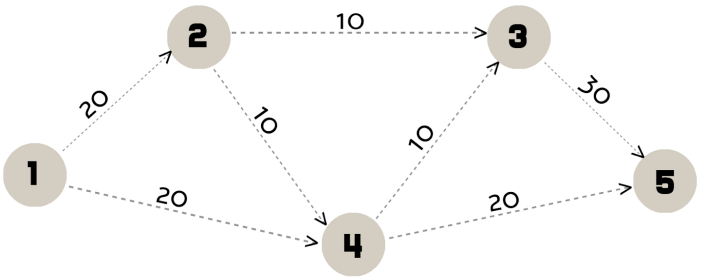
L'algorithme de parcours en largeur est un algorithme parcourant le graphe par couches. Les sommets de la couche n sont à une distance n du sommet source. L'exploration commence donc par la source, puis ses successeurs, puis les successeurs de ses successeurs, ... jusqu'à explorer le sommet puits, on sait alors que l'on peut s'arrêter. Grâce à un tableau répertoriant les prédécesseurs des sommets marqués (par convention, le sommet prédécesseur du sommet source est lui-même), on peut retrouver le chemin le plus court en nombre d'arcs menant d'un sommet à un autre (ici, menant du sommet source au sommet puits).

Voici le déroulement de l'algorithme de **DINIC** sur un réseau à 5 sommets et 7 arcs :

1. `Réseau :`

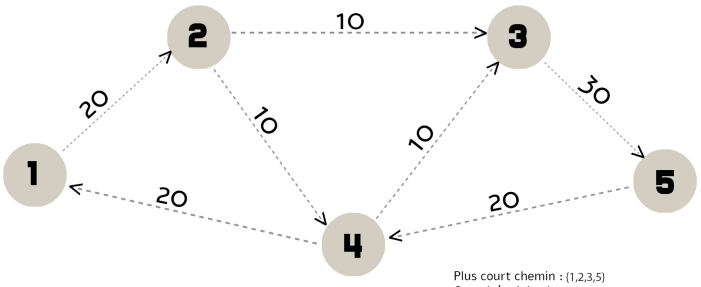


2. Graphe d'Écart :



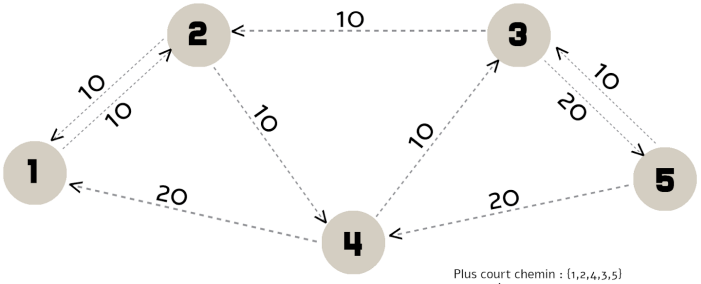
Plus court chemin : {1,4,5}
Capacité minimal : 20
Flot: 20

3. Graphe d'Écart :



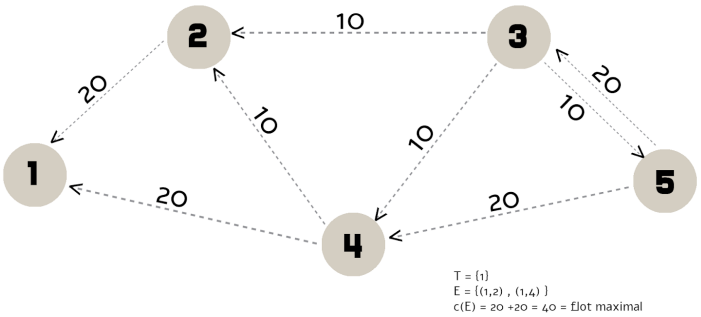
Plus court chemin : {1,2,3,5}
Capacité minimal : 20
Flot : 20

4. Graphe d'Écart :

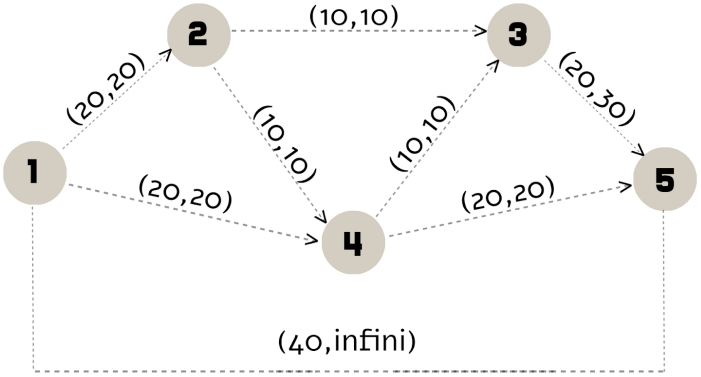


Plus court chemin : {1,2,4,3,5}
Capacité minimal : 10
Flot : 40

5. Graphe d'Écart :



6. Réseau :



Par convention, un graphe est constitué de n sommets et m arcs. On utilisera donc ces notations pour alléger le texte.

Pour représenter le réseau, on voit trois structures de données possibles. Elles seront illustrées ici par une représentation graphique basée sur l'exemple du réseau précédent, dans son état initial.

1. Une matrice d'incidence :

   Un tableau de tableaux : chaque case du premier tableau est elle-même un tableau. les sommets sont représentés en lignes et les arcs en colonnes. La case est à 0 si le sommet n'est pas incident à l'arc, 1 s'il est source de l'arc et -1 s'il est destination de l'arc.

	1→2	1→4	2→3	2→4	3→5	4→3	4→5
1	1	1	0	0	0	0	0
2	-1	0	1	1	0	0	0
3	0	0	-1	0	1	-1	0
4	0	-1	0	-1	0	1	1
5	0	0	0	0	-1	0	-1

2. Un tableau sommets - successeurs :

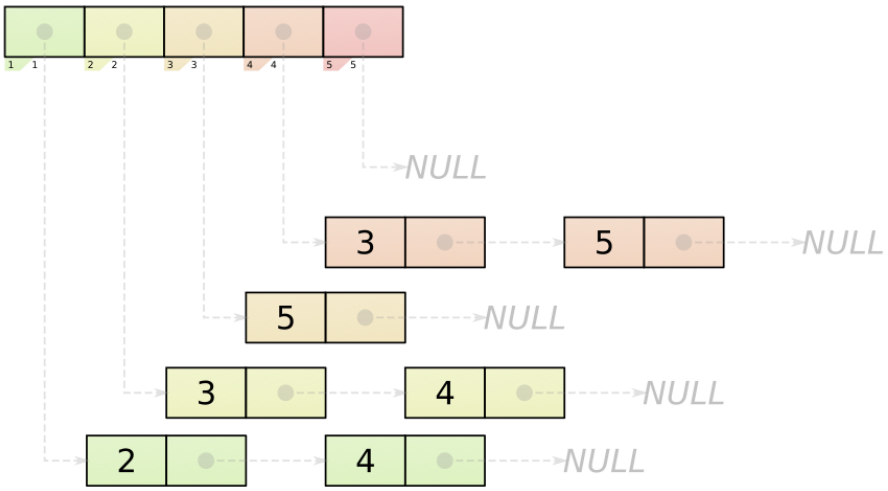
   Deux tableaux. Le premier tableau de la longueur du nombre d'arc est un tableau de pointeurs pointant vers le premier sommet successeur dans le second tableau. Le second tableau, de la longueur du nombre d'arcs, représente les successeurs des sommets du premier tableau. Les successeurs d'un même sommet sont contigus dans le second tableau.

1	3	5	6	8
1 1	2 2	3 3	4 4	5 5

2	4	3	4	5	3	5
1 1	2	2 3	4	3 5	4 6	7

3. Un tableau de listes de successeurs :

   Un tableau de listes de successeurs. Chaque case contient une liste. La tête de la liste indique quel est le sommet source et les maillons de la liste sont les successeurs de ce sommet source.



Comparons les avantages des trois structures de données sur 3 critères : le coût de stockage en mémoire, le coût de traitement en accès au successeur d'un sommet donné et finalement l'ajout et la suppression d'un successeur d'un sommet donné.

	Matrice d'incidence	Tableau sommets - successeurs	Tableau de listes de successeurs
Coût de stockage en mémoire	La dimension de la matrice est de $(n \times m)$. L'initialisation du tableau occupe une taille de $(n \times m)$, donc la complexité en coût mémoire vaut $(n \times m) \times sizeof(int)$.	La complexité en coût mémoire de cette structure de donnée est simplement de $(n + m)$.	Le tableau initial est de taille n et il y a m maillons répartis dans les différentes listes du tableau. Ces structures sont allouées dynamiquement. La complexité en coût mémoire est de nouveau de $(n + m)$.
Coût de traitement en accès au successeur	Il faut parcourir la matrice afin de trouver les successeurs d'un sommet vu que c'est une matrice d'incidence donc on ne peut pas se concentrer sur une seule ligne ou colonne. La complexité est donc de $O(n \times m)$.	On a directement l'indice dans le tableau de successeurs en fonction du tableau de sommets. Donc on n'a pas vraiment de traitement à effectuer. La complexité est donc de $O(n)$.	On a une liste chaînée donc il y a le coût de parcours des maillons. La complexité est donc de $O(n)$.
Ajout et suppression d'un successeur	Il faut redimensionner une matrice à 2 dimensions et donc réalouer plus ou moins de mémoire, et déplacer les données étant donné qu'elles sont stockées dans des cases contigues (principe du tableau). La complexité est en $O(n \times m)$.	Le second tableau, de successeurs, doit être redimensionné, donc il faut potentiellement déplacer le tableau. La complexité de cette opération est en $O(m)$.	L'ajout ou la suppression d'un maillon dans une liste a un coût constant et donc négligeable. Sa complexité est en $O(1)$.

Par ce tableau, on se rend compte que, pour le tableau sommets - successeurs et le tableau de listes de successeurs, leurs complexités sont les mêmes en stockage mémoire et en accès au successeur. Cependant, la deuxième structure de donnée à une complexité an ajout et suppression de successeurs plus faible que la première stucture de donnée. On abandonne donc l'idée d'implémenter la première structure de données. La matrice d'incidence à une complexité largement moins bonne que celle du tableau de listes de successeurs sur les différents critères. Pour résoudre le problème de maximisation de flot par l'algorithme de **DINIC**, nous allons donc utiliser la structure de données de tableau de listes de successeurs.

Pour réduir le nombre d'ajout et de suppressions d'arcs dans le graphe d'écart, lors de ses mises à jour entre deux itérations de **DINIC**, on pourrait, lors de la création du graphe à partir du réseau, créer pour chaque sommet dans le graphe un arc avec le flot égalant la capacité de l'arc du graphe et un arc inverse de flot nul. Ça permettrait d'utiliser les arcs déjà construits et de ne pas ajouter de nouvel arc ou d'en retirer un de flot nul.

Nous n'appliquerons pas cette idée car elle utilise beaucoup d'espace mémoire souvent inutil. L'espace économisé n'est pas nécessairement important, mais il devient significatif lorsque

le nombre d'arcs devient grand car on passe approximativement de m arcs à $2 \times m$ arcs. On notera que cette solution aurait pu être appliquée au tableau sommets - successeurs et il n'y aurait plus eu d'avantages à utiliser la soulution retenue plutôt que cette structure de données avec $2 \times m$ arcs pour les deux structures de données.

L'ordre d'apparition des successeurs dans la liste des successeurs n'importe pas, les structures de données ne tiendront donc pas compte de l'ordre alpha-numérique.

2.2 Implémentation de la structure de données choisie

Pour dérouler l'algorithme de **DINIC** et trouver le flot maximum du réseau donné, on utilisera 4 structures de données :

1. Une liste, et ses maillons, représentant les successeurs d'un sommet dans le réseau (listes constituant le tableau de listes de successeurs).
2. Une liste, et ses maillons, représentant les successeurs d'un sommet dans le graphe d'écart (listes constituant le tableau de listes de successeurs).
3. Une liste, et ses maillons, représentant les sommets constituant le plus court chemin en nombre d'arcs permettant d'améliorer le flot d'un graphe donné.
4. Une file permettant de trouver le plus court chemin en nombre d'arcs dans un graphe donné.

Voici le code C décrivant ces structures de données :

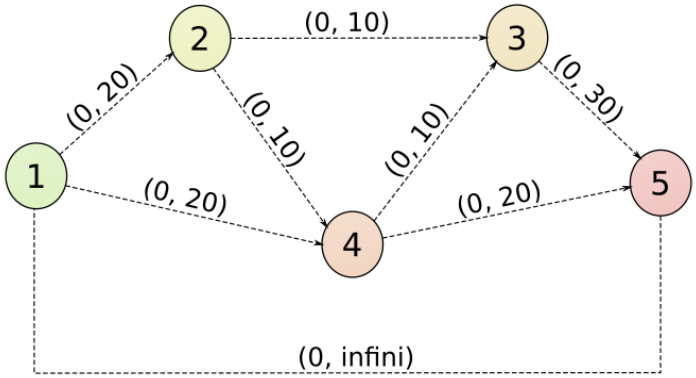
```
1         /* Reseau */
2
3         struct maillon_graph_reseau {
4             int id;
5             int flot;
6             int capacite;
7             struct maillon_graph_reseau* next;
8         };
9
10        struct liste_graph_reseau {
11            int id;
12            struct maillon_graph_reseau* head;
13        };
14
15
16
17        /* Graphe Ecart */
18
19        struct maillon_graph_ecart {
20            int id;
21            int flot_entrant;
22            struct maillon_graph_ecart* next;
23        };
24
25        struct liste_graph_ecart {
26            int id;
```

```
27              struct maillon_graph_ecart* head;
28          };
29
30
31
32          /* Chemin */
33
34          struct maillon_chemin {
35              int id;
36              int capacite_residual;
37              struct maillon_chemin * next;
38          };
39
40          struct liste_chemin {
41              struct maillon_chemin * head;
42          };
43
44
45
46          /* File */
47
48          struct file {
49              int* tab;
50              int taille;
51              int read_end;
52              int write_end;
53              int n;
54          };
```

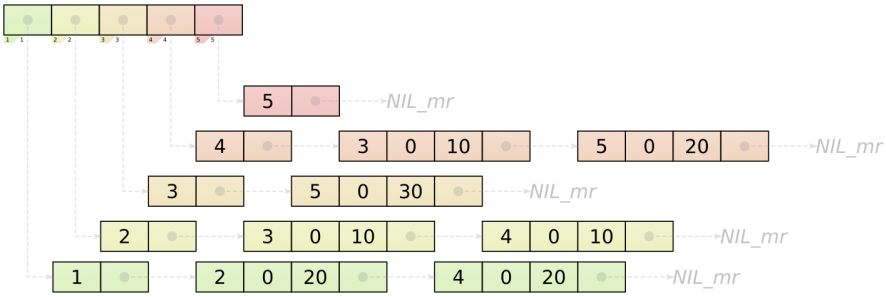
Lorsqu'un maillon est le dernier de la liste, il pointe vers un pointeur null de son type :

```
1           /* Reseau */
2
3           #define NIL_lr (struct liste_graph_reseau*) 0
4           #define NIL_mr (struct maillon_graph_reseau*) 0
5
6
7
8           /* Graphe Ecart */
9
10          #define NIL_lge (struct liste_graph_ecart *) 0
11          #define NIL_mge (struct maillon_graph_ecart *) 0
12
13
14
15          /* Chemin */
16
17          #define NIL_lc (struct liste_chemin *) 0
18          #define NIL_mc (struct maillon_chemin *) 0
```

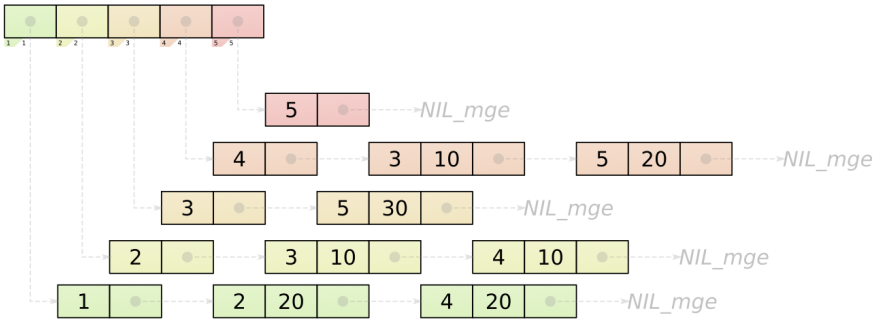
De manière schématique, sur un exemple donné ci-dessous, les structures de données ressembleraient donc à :



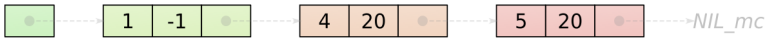
— Réseau : le réseau initial :



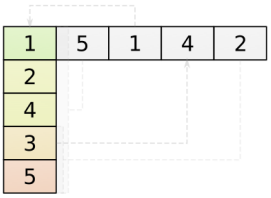
— Graphe Écart : le premier graphe d'écart construit à partir du réseau :



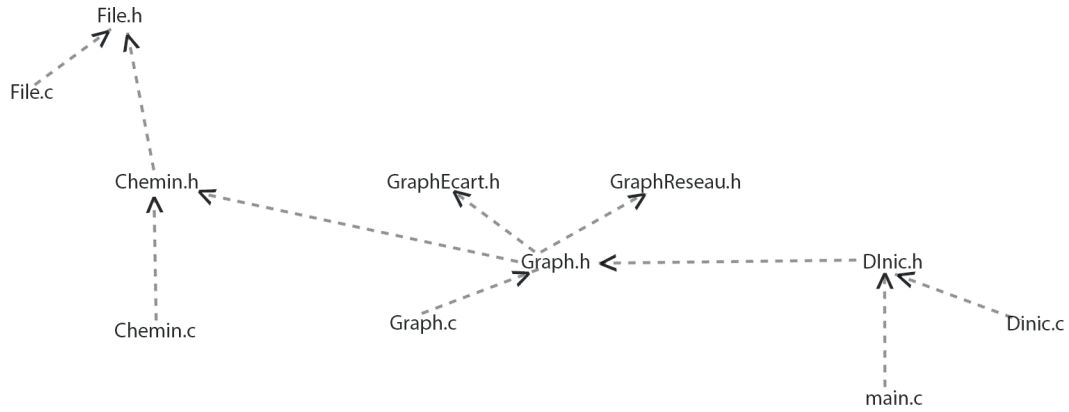
— Chemin : le premier chemin améliorant :



— File : la file à la fin de la construction du premier chemin améliorant :



L'arborescence des fichiers est la suivante :



La descritpion de la file ainsi que les déclarations des fonctions applicables dessus se trouvent dans le ficher `File.h` et ses fonctions sont décrites dans le fichier `File.c`.

Le réseau et le graphe d'écart sont tout deux des graphes et le chemin est construit à partir d'un graphe donc les fonctions dépendant de ces 3 structures de données sont contenues dans le ficher `Graph.c`. Ce dernier importe 4 fichiers d'entête : `GraphReseau.h`, `GraphEcart.h` et `Chemin.h` contenant la description des structures de données respectives. Le dernier fichier importé est `Graph.c`, contenant la déclaration des fonctions produites dans le fichier `Graph.c`.

Enfin, `Dinic.h` contient la signature de la fonction programmée dans le fichier `Dinic.c`, et `main.c` fait tourner le programme et maximise le flot du réseau donné en paramètres.

Ainsi, la File est importée dans le Chemin pour permettre de trouver le plus court chemin, le Chemin, le Graphe Écart et le Réseau sont importés dans le Graphe, le Graphe est importé dans

Dinic et enfin Dinic est importé dans le main.

`File.h` étant importé par tout les autres fichiers directement ou indirectement, toutes les inclusions de librairies sont faites dans ce fichier.

Chapitre 3

Mode d'emploi

L'entrée du programme est un fichier contenant la description d'un réseau au format **DIMACS**. (des exemples de ce format sont présents dans le dossier `./DIMACS/` ou encore sur `MOODLE`).

On va donc exécuter notre programme sur ce réseau pour maximiser le flot et obtenir un nouveau fichier $result.txt$ en sortie, contenant la description du réseau de flot maximum ainsi que le flot maximum final.

Pour lancer le programme, un `MAKEFILE` est mis à disposition pour gérer la compilation. La compilation se résume donc à lancer un terminal dans le répertoir courant et lancer la commande `make`. Cette commande exécute les commandes de compilation et liens avec les options `-Wall` et `-g`. On obtient ainsi tout les warnings et on peut lancer le programme sous un débogueur comme `gdb`. Le compilateur utilisé est `gcc`. L'exécutable produit est `a.out`.

Le programme nécessite un fichier en entrée, fournissant le réseau initial. Il faut saisir les chemin et le nom du fichier en question. On peut donc exécuter les lignes suivantes :

```
1    make
```

Commande compilant et faisant les liens d'import. Un fichier 'a.out' exécutable est généré. Le programme peut être lancé par exemple des différentes manières suivantes :

```
1    ./a.out
2    DIMACS/net1.txt
```

Le fichier 'result.txt' contient à présent le réseau à 5 sommets de flot maximum.

```
1    ./a.out
2    DIMACS/net2.txt
```

Le fichier 'result.txt' contient à présent le réseau à 6 sommets de flot maximum.

```
1    ./a.out
2    DIMACS/G_100_300.max
```

Le fichier 'result.txt' contient à présent le réseau à 102 sommets de flot maximum.

```
1    ./a.out
2    DIMACS/G_900_2700.max
```

Le fichier 'result.txt' contient à présent le réseau à 902 sommets de flot maximum.

```
./a.out
DIMACS/G_2500_7500.max
```

Le fichier ‘result.txt‘ contient à présent le réseau à 2502 sommets de flot maximum.

Les fonctions `buildGraph`, `buildRG`, `shortestPath`, `minCapa`, `updateFlowInRG`, `updateFlowInNet`, `dinic` et `main` sont décrites en pseudo-code dans un autre fichier.

Chapitre 4

Description des exemples traités

Tous les fichiers de graphes fournis se trouvent actuellement dans le répertoire **DIMACS**.
Tous les fichiers ayant été donnés ont été traités. Les résultats obtenus sont les suivants :
— net1.txt : flot maximum = 40.
— net2.txt : flot maximum = 15.
— G_100_300.max : flot maximum = 9860177.
— G_900_2700.max : flot maximum = 28258807.
— G_2500_7500.max : flot maximum = 42791871.
Les résultats sont bien ceux souhaités.

Conclusion

Le projet nous a permis de consolider nos connaissances en C, structures de données, algorithmes de parcours de graphes et maximisation de flot dans un réseau.

Nos principales difficultés venaient d'une erreur d'inattention.

Cependant, en travaillant à deux et en demandant un peu d'aide à un enseignant, nous avons finis par résoudre cette erreur et notre programme fonctionne pour tout réseau à maximiser et des fichiers ayant un format pas nécessairement strictement identiques.

Chapitre 5

Améliorations Possibles :

On remarque qu'au niveau de la structuration des structres, on peut améliorer et garder une structure globale pour un sommet et contruire des graphs différents en fonction de cela.

On pense également que notre structure est plus claire et compréhensible que d'autres structures moins coteuses en termes de mémoire.

Mais au niveau du traîement, nous avons essayés de ne pas faire de parcours inutil et nous avons ajoutés des conditions pour éviter des parcours inutiles.

Chapitre 6

Technique :

Nous avons utilisés Git pour gérer notre projet, ce qui nous a permis de travailler plus efficacement. Techiniquement, on a pas eu de contraintes.

Bilan personnel

On tient à remercier Monsieur Julien Forget de nous avoir données des pistes pour corriger des erreurs. À travers ce projet, nous avons améliorés nos conaissances sur les poiteurs et le parcours des graphes. Nous avons également enrichis nos compétences en programmation C avancé et la compréhension des algorithmes de parcours.

On a essayés de créer des tableaux dynamique qui nous permettent de faire fonctionner l'algorithme sur n'importe quel ficher de n'importe quelle taille de graphe.

Notre méthode de travail était efficace étant donné que nous avons tous les deux essayés de répondre au probleme séparement et avons réalisés le déboguing ensemble. Nous n'avons pas répartis les tâches, nous avons répartis les algorithmes à programmer. Cependant, le déboguing a été fait ensemble, ce qui nous a fait gagner beaucoup de temps.