CRO TP3 : Graphes

une séance sur machine 4h, 6 octobre 2025

1 Introduction

Nous allons travailler sur les graphes orientés. Il y a deux représentations statiques utilisées pour un graphe orienté $G = (X, A)$, avec X ensemble de sommets et A ensemble des arcs entre les sommets :

— Matrice d'incidence : une ligne par sommet, une colonne par arête

— Matrice d'adjacence : une ligne par sommet, une colonne par sommet (matrice symétrique pour les graphes non orientés). par exemple l'élément de la matrice `M[i][j]` vaut 1, cela signifie qu'il y a un arc du sommet `i` au sommet `j` (les sommets sont donc numérotés).

La complexité peut rapidement être un problème lorsque l'on traite des graphes de tailles importantes, ce qui est souvent le cas en réseau. Théoriquement on peut avoir un nombre d'arcs ($|A|$) de l'ordre de $|X|^2$, mais en général $|X|^2 \gg |A|$. Du fait de la taille des structures classiques, on utilise souvent des représentations dynamiques :

— Structures auto-référencées (généralisation des structures d'arbres, avec des liste de voisins par noeuds), assez difficile à manipuler.

— Liste de sommets suivants

Dans ce TP de quatre heures, nous utiliserons la representation statique utilisant une matrice d'adjacence. Le type proposé pour un graphe est le suivante :

```
struct graph {
  int nb_sommets;
  int **adj;
};
typedef struct graph GRAPHE;
```

QUESTION 1 ▶ Le champ `adj` représente la matrice d'adjacence. L'entrée `adj[i][j]` représente l'existence ou pas d'un arc entre le noeud `i` et le noeud `j`. Pouquoi n'avons nous pas utilisé la déclaration du champs sous la forme suivante : `int adj[N][N]` pour cette matrice ?

2 Prise en main du code (matrice d'adjacence)

Le problème de la manipulation de graphes en C, c'est qu'il n'existe pas de bibliothèque standard permettant de manipuler les graphes (affichage, ajout de noeud etc.). Comme pour les autres structures algorithmiques d'ailleurs, en C nous devons tout faire à la main.
Depuis quelques années le langage DOT, langage de description de graphes dans un format textuel, est supporté par l'ensemble d'outils open source Graphviz [1] (voir ici pour l'installation sur votre machine : `https://graphviz.gitlab.io/download/`). Nous vous proposons quelques fonctions utiles pour étudier les algorithmes de graphe en C (comme pour le sudoku : lire une matrice d'adjacence, l'écrire ou visualiser le graphe qu'elle représente avec l'outil `dotty`).
Si vous n'avez pas dotty installé sur votre machine, vous ne pourrez pas visualiser les graphes, ce n'est pas grave, vous pourrez toujours travailler mais en visualisant uniquement les matrices d'adjacences (vous aurez simplement un message d'erreur concernant le fait que `dotty` n'est pas installé si vous utilisez la fonction `dotty_adj_dot` présentée plus loin).
Attention, il peut y avoir des problèmes lors de l'installation de dotty (graphviz) sous linux :
(`https://bugs.launchpad.net/ubuntu/+source/graphviz/+bug/1016777`)
il faut installer les polices `xfonts-100dpi` et redémarrer le serveur X :

1. `https://fr.wikipedia.org/wiki/DOT_(langage)`

```
sudo apt-get install xfonts-100dpi
restart X session (log out and back in, for example)
```

Les fonctions qui sont à votre disposition (dans le fichier `util_adj.c`) sont les suivantes :

```
/* création et initialisation à 0 d'un graphe de                */
/* nb_sommets sommets: aucun arcs                               */
GRAPHE init_adj_mat(int nb_sommets);

/* lecture d'une matrice d'adjacence dans un fichier,           */
creation du graphe correspondant */
GRAPHE read_adj_mat(char *nom_fich);

/* écriture d'une matrice d'adjacence dans un fichier          */
 void affiche_adj_mat(FILE *fich,GRAPHE g);

/* affichage d'un graphe au format dot                          */
/* a partir du format matrice d'adjacence                       */
void print_adj_dot(FILE *fich,GRAPHE g); //à l'écran
void affiche_adj_dot(char *nom_fich,GRAPHE g); //dans un fichier
void dotty_adj_dot(GRAPHE g);  //appelle  dotty, si il est installé
```

QUESTION 2 ► Téléchargez l'archive présente sur Moodle, décompressez là, et parcourez le fichier `main.c`. Regardez le main afin de comprendre ce qu'il fait. Sur la Figure 1, vous avez la représentation textuelle de la matrice d'adjacence correspondant au fichier `graph_ex1.mat` ainsi que sa représentation `dotty` grâce à la fonction `dotty_adj_dot` qui vous est fournie dans `util_adj.c`

QUESTION 3 ► Vérifiez que vous savez utiliser les fonction `read_adj_mat`, `affiche_adj_mat` et (dans le cas où `dotty` est installé) `dotty_adj_dot`. Cela vous vous permettra de débugger vos programmes.

```
0 1 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 1 0 1 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
```

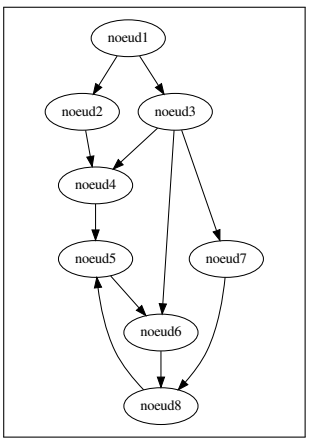

FIGURE 1 – Matrice d'adjacence du premier exercice et sa représentation avec `dot`.

3 Fermeture Transistive

la fermeture transitive R^* d'une relation R est telle que

$$xR^*y \Leftrightarrow \exists z_1, ..., z_n / xRz_1, z_1Rz_2, ..., z_nRy$$

Lorsque l'on évoque la fermeture transitive de graphes, la relation considérée est "l'adjacence" dans un graphe orienté G. Le calcul de la fermeture transitive permet de répondre aux questions concernant l'existence de chemins entre x et y dans un graphe G et ceci pour tout couple de sommets (x, y).

Algorithme de la fermeture transistive

— On modifie au fur et à mesure la matrice d'adjacence de G (M).

— Répéter la procédure `iterationFT` jusqu'à stabilisation.

— iterationFT(MATRICE M)

```
DEBUT
    POUR i VARIANT de 1 à N
        POUR j VARIANT de 1 à N
            SI M[i,j]=1 ALORS
                POUR k VARIANT de 1 à N
                    SI M[j,k]=1 ALORS M[i,k]=1
                FINPOUR
            FINSI
        FINPOUR
    FINPOUR
FIN
```

— On repète cette procédure jusqu'à ce qu'aucune valeur de la matrice ne change.

La solution que vous devez obtenir est représentée en figure 2

```
0  1  1  1  1  1  1  1
0  0  0  1  1  1  0  1
0  0  0  1  1  1  1  1
0  0  0  0  1  1  0  1
0  0  0  0  1  1  0  1
0  0  0  0  1  1  0  1
0  0  0  0  1  1  0  1
0  0  0  0  1  1  0  1
```

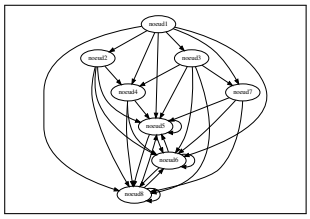

FIGURE 2 – Fermeture transitive du graphe de la figure 1

QUESTION 4 ▶ Créez un fichier `graphe.c` dans lequel vous programmerez la fonction de fermeture transitive. Modifier le `Makefile` pour qu'il prenne en compte le fichier `graphe.o` dans la production de l'exécutable `main`.

QUESTION 5 ▶ Programmez d'abord, dans ce fichier `graphe.c`, une fonction `int iterationFT(GRAPHE g2)` qui implemente la procédure ci-dessus sur un graphe `g2` (i.e. qui modifie `g2`) et qui renvoie un entier qui vaut 1 si le graphe `g2` a été modifié et 0 si g2 n'a pas été modifé.

QUESTION 6 ▶ Programmez maintenant une fonction qui calcule la fermeture transistive d'un graphe orienté en utilisant la procédure précédente et appliquez là au graph du fichier `graph_ex1.mat`.

3.1 Calcul du plus court chemin : algorithme de Dijkstra

Lorsque les arcs des graphes sont valués (i.e. chaque arc à un 'poids' que nous supposerons entier), on peut calculer le *plus court chemin* entre deux noeuds, qui est le chemin dont le poids cumulé sur les arcs est minimum.
Pour manipuler les graphes valués, nous allons avoir besoin d'une autre structure de données. En effet, si nous utilisons la matrice `adj` pour représenter les poids, nous ne pourrons pas distinguer le cas d'un arc de poids 0 du cas ou il n'y a pas d'arc. Nous vous proposons donc une extension du type `GRAPHE` avec le type `GRAPHE_POIDS` :

```
struct graphe_poids {
  int nb_sommets;
  int **adj;
  int **poids;
  int *marque;
};
typedef struct graphe_poids GRAPHE_POIDS;
```

Le matrice `adj` joue le même rôle qu'avant (existence ou non d'un arc entre deux sommets) et la matrice `poids` contriendra le poids présent sur chaque arc. Le champ `marque` permettra de 'marquer' les sommets au cours de l'algorithme.
Un algorithme celèbre pour calculer le plus court chemin est l'algorithme de Dijkstra, qui fonctionne pour une pondération positive des arcs et qui fonctionne avec un marquage des noeuds (d'ou le champ `marque` dans le type ci-dessus).
De nouveau, nous proposons les mêmes fonctions (dans le fichier `util_poids.c`) pour travailler sur les algorithmes de graphes valués en C :

```
GRAPHE_POIDS init_poids_mat(int nb_sommets);
GRAPHE_POIDS read_poids_mat(char *nom_fich);
void affiche_poids_mat(FILE *fich,GRAPHE_POIDS g);
void print_poids_dot(FILE *fich,GRAPHE_POIDS g);
void affiche_poids_dot(char *nom_fich,GRAPHE_POIDS g);
void dotty_poids_dot(GRAPHE_POIDS g);
```

Rappel de l'algorithme de Dijkstra L'algorithme de Dijkstra [2] permet de calculer dans un graphe $G = (X, A)$, la distance entre tous les sommets du graphe et un sommet particulier, appelé *sommet source* et noté s. L'algorithme complet utilise une file à priorité que nous n'étudierons ici, nous implémenterons donc une version simplifiée de l'algorithme de Dijkstra.
On utilise un vecteur de distance $dist$. l'entrée $dist_i(x)$ correspond, au cours de l'algorithme, à la distance entre le sommet s et le sommet x *à l'étape* i (une étape est le point 2. de l'algorithme ci-dessous). On note $\Gamma^-(x)$ l'ensemble des prédécesseurs de x (c'est à dire, l'ensemble des sommets qui ont un arc vers x) et $P(yx)$ le poids de l'arc $y \to x$. Le tableau $dist$ est rempli successivement avec les chemins de longueur 2 qui partent de s, puis les chemins de longueur 3, etc.
Algorithme de Dijkstra :

1. Chaque sommet x est initialisé à une distance infinie de s (sauf s lui-même bien sûr) : $\forall x \in X, x \neq s\ dist_0(x) = \infty$, et $dist_0(s) = 0$ et s est marqué.
2. On réévalue $dist$ pour les x non marqués :
   — $\forall x \in X$, x non marqué,
   $$dist_i(x) = Min(dist_{i-1}(x), Min_{y \in \Gamma^-(x)}(dist_{i-1}(y) + P(yx)))$$
   — Le sommet x de distance $dist_i(x)$ minimum est marqué
3. On recommence l'étape 2. jusqu'à ce que tous les sommets soient marqués

Considérons le graphe de la figure 3, vous le trouverez dans le fichier `poids_ex1.mat`, vous pouvez donc créer le graphe valué de la figure 3 et l'afficher avec `dotty` avec la séquence :

```
GRAPHE_POIDS g3;
g3=read_poids_mat("poids_ex1.mat");
dotty_poids_dot(g3);
```

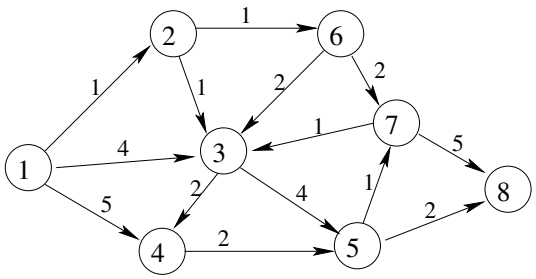

FIGURE 3 – Graphe pour l'exercice sur l'algorithme de Dijkstra

1. Appliquer l'algorithme de Dijkstra à partir du sommet 1.

2. https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

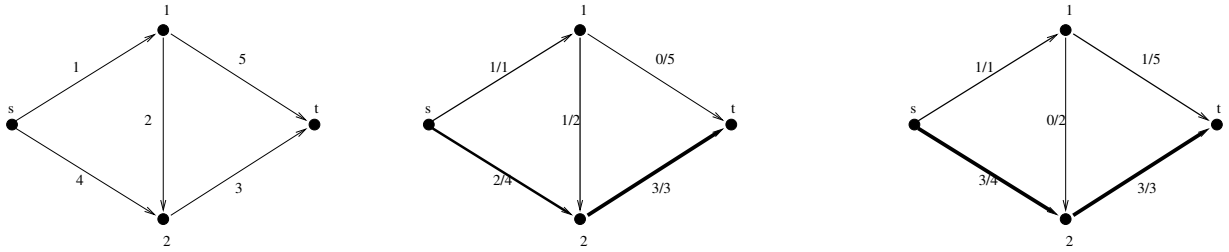

FIGURE 4 – Exemple simple : à gauche un graphe avec des capacités, a droite deux flot admissibles conservatifs allant de `s` à `t`. Un de ces flot est meilleur que l'autre comprenez-vous pourquoi ?

4 Pour ceux qui ont terminé : calcul de flot dans les graphes (Algorithme de Ford-Fulkerson)

Le problème du flot maximum consiste à trouver la meilleure répartition d'un flot (pensez à un flot de voitures sur les routes ou à du pétrole dans des pipelines) sur les arcs d'un graphe qui ont une certaines capacité.
C'est un problème très compliqué, il peut arriver par exemple que l'on ferme une voie temporairement pour l'agrandir parce qu'elle était tout le temps embouteillée et que sa fermeture améliore la circulation dans toute la ville.
Concernant la formalisation sous forme de graphe, on dispose maintenant de graphes dont les sommets sont labélisés par un numéro (entier), et les arcs sont labélisés par une capacité (entière pour simplifier). Cette capacité de l'arc resprésente le maximum du flot que peut supporter l'arc (pensez en litres par seconde – diamètre du tuyau – ou en voitures par minutes). Un flot sur un tel graphe consiste en une deuxième labélisation de chaque arc qui est la valeur effective du flot sur cet arc. La formalisation de ce problème est la suivante :

— On dispose d'un graphe $G(V, A)$ orienté
— Deux sommets particuliers sont identifiés : s appelé *source* et t appelé puits.
— Les arcs de G sont étiquetés par une valeur entière positive appelée *capacité*, $c : A \mapsto N^+$
— Trouver une fonction $f : A \mapsto N^+$ qui soit un *flot admissible conservatif maximal*

Avec les définitions suivante pour un flot :

— admissible : respecte la capacité des arcs.
— Conservatif : pour tous les nœuds, sauf la source le puit, le flot entrant est égal au flot sortant.
— Maximal : qui maximise la capacité sortant de s.

QUESTION 7 ▶ Comprenez l'exemple de la figure 4. On voit bien que l'algorithme qui consiste à remplir de manière gloutone les capacités dès que l'on peut n'est pas un bon algorithme.
L'algorithme de **Ford-Fulkerson** [3] consiste partir d'une flot admissible et de trouver des *chaînes améliorantes*. Il est le suivant :

— Trouver un flot admissible (par exemple le flot nul)
— Essayer d'améliorer ce flot tant que c'est possible
— Pour améliorer ce flot, on trouve une *chaîne améliorante* : une chaîne entre s à t sur laquelle la quantité de flot peut être augmentée.
  — On marque progressivement les sommets en commençant par s
  — On marque les successeurs y des sommets x marqués si $f(x, y) < c(x, y)$ (i.e. on peut augmenter le flot de x vers y)
  — On marque les prédécesseurs y des sommets x marqués si $f(y, x) > 0$ (i.e. on peut augmenter le flot de x vers y)
  — La marque identifie par quels successeur (ou quel prédécesseur) la chaîne améliorante passe.

3. `https://fr.wikipedia.org/wiki/Algorithme_de_Ford-Fulkerson`

— Une fois la chaîne trouvé, on augmente le flot de 1 sur le chemin trouvé (augmente le flot sur les arcs arrivant sur un noeud labélisé +x, diminue le flot sur les arcs arrivant sur les noeud labélisé -x)

Ou en langage algorithmique :

On marque l'entrée du réseau par $*$
TANT QUE non stable
    POUR CHAQUE x marqué
        POUR CHAQUE y successeur de x non marqué
            SI $f(x, y) < c(x, y)$ ALORS marquer(y) par (+x)
        FINPOUR
        POUR CHAQUE y prédécesseur de x non marqué
            SI $f(y, x) > 0$ ALORS marquer(y) par (-x)
        FINPOUR
    FINPOUR
    SI t est marqué ALORS
        on a trouvé une chaîne améliorante
FIN TANTQUE

QUESTION 8 ▶ Programmez l'algorithme de Ford-Fulkerson avec la structure de données ci-dessous (des fonctions vous sont fournies sur Moodle ainsi qu'un exemple de graphe).

```
struct graphe_flot {
  int nb_sommets;
  int **adj;
  int **poids;
  int **flot;
  int *marque;
};
typedef struct graphe_flot GRAPHE_FLOT;

#endif
```

FIGURE 5 – Type proposé pour une représentation de graphe pour l'algorithme de calcul des flot maximum.