

Structures de données et algorithmes II

Devoir de TP libre

2023 - 2024

Cadre du problème : graphe d'information relationnelle

Dans le cadre d'un outil d'aide à l'enquête criminelle, on se propose de modéliser des relations entre « entités ». Ces entités sont des objets, des lieux et des individus. La figure 1 ci-dessous présente graphiquement l'information relationnelle que l'on souhaite modéliser.

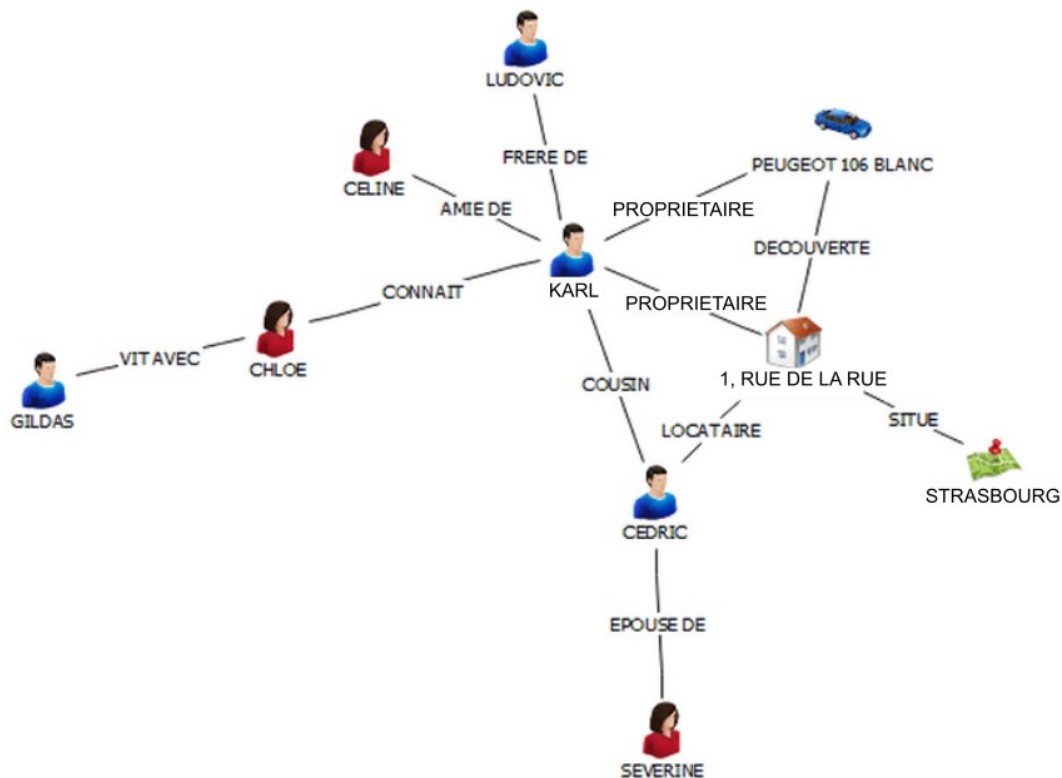


FIGURE 1 – Ensemble des relations que l'on souhaite modéliser.

Chaque entité a un nom (chaîne de caractères) et un type. Le nom de l'entité est unique. On se limite à quatre types différents d'entités : Personne, Objet, Adresse et Ville. Un nom NULL est utilisé comme Ω . L'ensemble des entités qui sont en relation forme un graphe. Chaque relation est également étiquetée (voir figure 1). Il ne peut y avoir qu'une seule relation entre deux mêmes entités, si bien qu'il s'agit d'un graphe *simple*. Par ailleurs, le graphe est *non-orienté* et il est *valué*.

Les types de relations sont définies sous la forme d'une énumération :

1	Lien
2	frère/sœur
3	cousin/cousine
4	père/mère
5	oncle/tante
6	époux/épouse
7	ami de
8	vit avec
9	connait
10	supérieur de
11	collègue de
12	locataire
13	travaille à
14	propriétaire
15	situé à
16	découvert à

Chaque type de relation correspond alors à un numéro, comme ici : 2 signifie « frère ou sœur de », 15 signifie « propriétaire » :

```
typedef enum { FRERE=2, COUSIN, PARENT,
ONCLE, EPOUX, AMI, VIT, CONNAIT, CHEF,
COLLEAGUE, LOCATAIRE, TRAVAILLE,
PROPRIETAIRE, SITUE, DECOUVERT } rtype;
```

Notez que l'on commence l'énumération à 2 ! Par convention, la valeur 0 correspond à Ω . La valeur 1 correspond à une relation **inconnue**. Cette dernière signifie, qu'on ne sait pas si les deux entités ont une relation ou non.

1. Opérations de classement des relations (2,5pts)

1.1 Définir des opérations qui permettent de classer les relations entre paires de personnes en fonction de l'identifiant :

```
bool est_lien_parente(rtype id);
bool est_lien_professionel(rtype id);
bool est_lien_connaissance(rtype id);
```

Pour le premier, il s'agit des relations de 2 à 6, pour le second des relations 10 à 11 et pour le dernier des relations 7 à 9. Par exemple *est_lien_parente(ONCLE)* renvoie *vrai* alors que *est_lien_parente(CONNAIT)* renvoie *faux*.

1.2 Ecrire une opération qui permet de renvoyer une chaîne de caractère correspondant au type de relation :

```
char* toStringRelation (rtype id);
```

Par exemple *toStringRelation(FRERE)* renvoie la chaîne de caractères « frère ou sœur de ». Utiliser pour cela une table statique par adressage associatif.

2. Liste d'adjacence (3pts)

Le graphe est encodé sous la forme *d'une liste chaînée d'adjacences*. Pour cela nous définissons d'abord un type « liste de pointeurs void * » :

```
typedef struct s_node {
    void *val; // pointeur quelconque
    struct s_node *suiv;
} *listeg;
```

Contrairement à la liste générique vue jusqu'ici (en TD et en TP), il s'agit d'une liste de pointeurs seulement. Il n'est pas nécessaire de cloner les objets lorsqu'ils sont ajoutés. On se contente de copier le pointeur. Seule la recherche nécessite encore un pointeur de fonction. Ecrire les opérations sur cette liste :

```

listeg listegnouv() ;
listeg adjtete(listeg lst, void *x) ;
listeg adjqueue(listeg lst, void *x) ;
listeg suptete(listeg lst) ;
listeg rech(listeg lst, void *x, int(*comp)(void *, void *)) ;
void *tete(listeg lst) ;
int longueur(listeg lst) ;
bool estvide(listeg lst) ;
void detruire(listeg lst) ;

```

Attention : la fonction **rech** crée une nouvelle **listeg** qui contient **toutes** les références pour lesquelles la fonction **comp** renvoie la valeur 0. Là encore, on ne fait pas de clonage. L'ordre dans lequel les éléments sont placés dans la **listeg** résultat n'a pas d'importance.

3. Construction du graphe (5,5pts)

3.1 On définit maintenant une structure de données pour représenter toute l'information relationnelle de la figure 1. Il s'agit de définir un type de données **Relations** qui modélise le graphe. Pour cela, on définit d'abord un type **Entite** :

```

typedef enum { PERSONNE, OBJET, ADRESSE, VILLE } etype;
typedef struct s_entite {
    char nom[LONG_NOM_MAX]; // le nom de l'entité p.ex « Peugeot 106 »
    etype ident; // l'identifiant associé, p.ex OBJET
} *Entite;

```

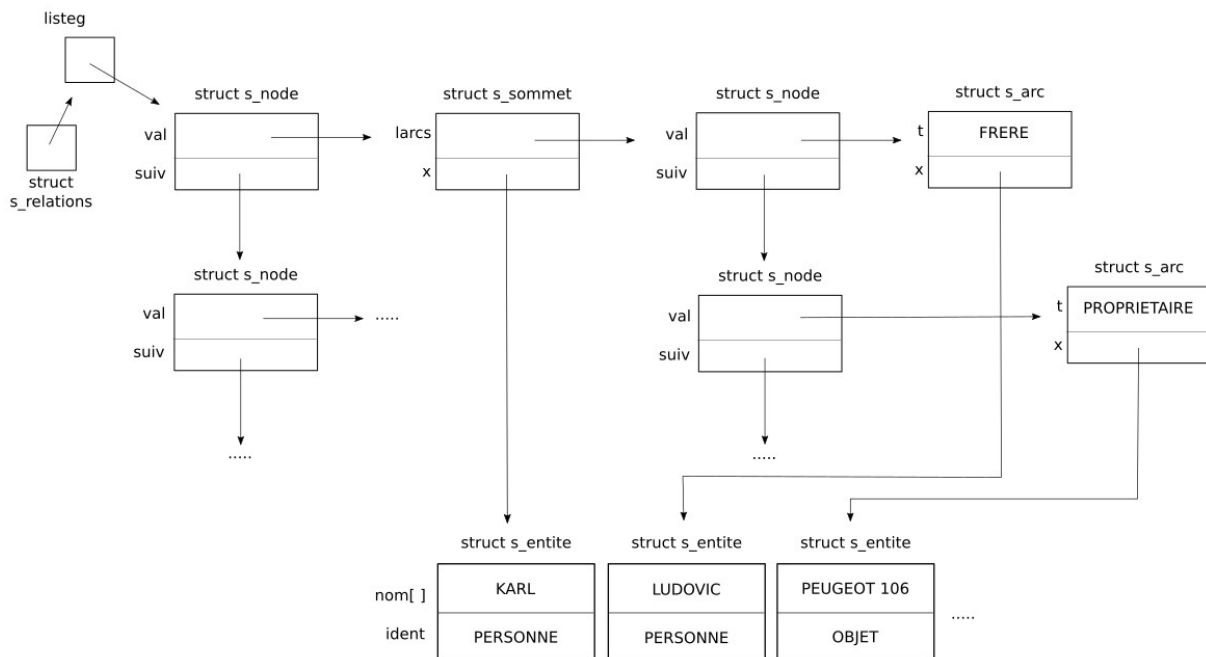


FIGURE 2 – Structure **Relations** avec les chaînages en mémoire. On ne montre ici qu'une petite partie de l'exemple de la figure 1.

Puis, on définit des structures pour les sommets et les arcs de graphe :

```

typedef struct s_sommet {
    // A DEFINIR
} *Sommet;

```

```

typedef struct s_arc {

```

```

        // A DEFINIR
    } *Arc;

```

Et enfin, on définit la structure **Relations**.

```

typedef struct s_relations {
    // A DEFINIR
} *Relations;

```

Une représentation graphique de cette structure, telle qu'elle est en mémoire, est donnée par la figure 2. Se servir de cette figure pour compléter les structures **Sommet**, **Arc** et **Relations**.

3.2 Ecrire les constructeurs de base de **Sommet**, **Arc** et **Relations**, ainsi qu'éventuellement leur destructeur (libération de la mémoire). Pour **Relations**, on utilisera une programmation dynamique par mutation :

```

Entite creerEntite(char *s, etype e) ;
Sommet nouvSommet(Entite e) ;
Arc nouvArc(Entite e, rtype type) ;
void relationInit(Relations *g);
void relationFree(Relations *g);

```

La destruction des **Sommet** et **Arc** se fera au niveau de la destruction de **Relations**. C'est pour cela qu'on ne définit pas de destructeurs explicites pour ces deux structures de données.

3.3 Ecrire des opérations de comparaison.

```

int compEntite(void *e, void *string) ;
int compSommet(void *s, void *string);
int compArc(void *a, void *string) ;

```

Pour ces fonctions, le premier paramètre void * correspond à la structure, respectivement **Entite**, **Sommet** et **Arc** et le second à une chaîne de caractères (char *). On utilisera *strcmp*.

3.4 Ecrire les deux constructeurs qui permettent d'ajouter une entité et une relation au graphe.

```

void adjEntite(Relations g, char *nom, etype t);

```

Cette fonction vérifie qu'il n'existe pas déjà une entité portant le même nom. En effet, le nom de l'entité doit rester unique dans le graphe. Ici, **g** est modifié par effet de bord.

```

// PRE: id doit être cohérent avec les types des sommets correspondants à x et y
//      p.ex si x est de type OBJET, id ne peut pas être une relation de parente
// PRE: strcmp(nom1,nom2)!=0
void adjRelation(Relations g, char *nom1, char *nom2, rtype id);

```

Remarque : dans le cas où l'arc existe déjà, son « type » est mis-à-jour et remplacé par *id*.

4. Exploration des relations de connaissances entre personnes (5pts)

4.1 Ecrire des fonctions permettant de créer des listes de relations avec une entité de nom x :

```

listeg en_relation(Relations g, char *x) ;

```

La *listeg* renvoyée doit donner toutes les arêtes de x (le void * de *listeg* représente le type *Arc*). Ici on ne crée pas de nouvelle liste. On renvoie simplement la liste *lars* du *Sommet* correspondant à x (voir figure 2). On utilisera la fonction *rech*.

```
listeg chemin2(Relations g, char *x, char *y) ;
```

La *listeg* renvoyée doit donner toutes les entités z telles que x a une relation avec z et z a une relation avec y (le void * de *listeg* représente ici le type *Entite*), quelle que soit ces relations et quelles que soient les entités. z doit être différent de x et de y. Une nouvelle liste est créée par cette fonction. *chemin2* permet donc d'obtenir tous les chemins de x à y de longueur 2 dans le graphe des relations.

4.2 Ecrire une fonction qui vérifie si deux personnes ont un lien de parenté / connaissance :

```
// PRE : strcmp(x,y)!=0
bool ont_lien_parente(Relations g, char *x, char *y);
```

Dans l'exemple, si x est Karl et y est Ludovic alors la fonction renvoie vrai. Mais si y est Céline alors la fonction renvoie faux. On utilisera *est_lien_parente*.

4.3 Ecrire une fonction permettant de tester si deux individus x et y se connaissent.

On suppose que x et y se connaissent s'ils ont un lien de parenté, un lien professionnel ou un lien de connaissance. On suppose par ailleurs que x et y se connaissent s'il ont tous les deux un lien de parenté avec un même individu z.

```
// PRE: les sommets correspondants à x et y sont de type PERSONNE
bool se_connaissent(Relations g, char *x, char *y) ;
```

Dans l'exemple, si x est Karl et y est Ludovic alors la fonction renvoie vrai. Si y est Céline alors la fonction renvoie vrai. Si x est Céline et y est Chloé alors la fonction renvoie faux. Si x est Ludovic et y est Cédric alors la fonction renvoie vrai car ils ont tous les deux un lien de parenté avec Karl.

4.4 Ecrire une fonction permettant de tester si deux individus se connaissent très **probablement**.

On suppose que x et y se connaissent très probablement s'ils ont au moins une connaissance z commune (qui est une personne), mais pas de connaissance directe et que l'une des deux connaissances au moins est une relation de parenté, mais pas les deux à la fois (ou exclusif).

```
// PRE: les sommets correspondants à x et y sont de type PERSONNE
bool se_connaissent_proba(Relations g, char *x, char *y) ;
```

Dans l'exemple, si x est Karl et y est Ludovic alors la fonction renvoie faux. Si y est Céline alors la fonction renvoie faux. Si x est Céline et y est Ludovic alors la fonction renvoie vrai, car ils connaissent Karl qui a un lien de parenté avec Ludovic. Si y est Chloé alors la fonction renvoie faux car même si elles connaissent toutes les deux Karl, il n'y a pas de lien de parenté.

4.5 Ecrire une fonction permettant de tester si deux individus se connaissent **peut-être**. On suppose qu'ils se connaissent peut-être s'ils ont au moins une connaissance commune, mais pas de connaissance directe ou très probablement.

```
bool se_connaissent_peutetre(Relations g, char *x, char *y) ;
```

Dans l'exemple, si x est Karl et y est Ludovic alors la fonction renvoie faux. Si y est Céline alors la fonction renvoie faux. Si x est Céline et y est Ludovic alors la fonction renvoie faux, car ils se connaissent probablement. Si y est Chloé alors la fonction renvoie vrai car elles connaissent toutes les deux Karl.

5. Affichages (2pts)

Ecrire les opérations d’affichage suivantes :

```
void afficheIg(listeg l, void(*aff)(void *));  
void afficheEntite(void *x) ;  
void afficheArc(void *x) ;
```

afficheEntite affiche le nom et le type, sous la forme : nom :type, par ex. KARL :personne

afficheArc affiche sous la forme : -- rtype--> nom :type, par ex. --connait-->CHLOE:personne

6. Parcours des relations (2pts)

6.1 Ecrire une fonction permettant d’afficher tous les degrés de relation avec une entité de nom x.

```
void affiche_degre_relations(Relations r, char *x);
```

L’affichage précisera le degré de relation : 1 relation directe, 2 relation par un chemin de longueur 2, etc. Par exemple si x est Chloe, on affichera :

```
CHLOE  
-- 1  
  GILDAS  
  KARL  
-- 2  
  1, RUE DE LA RUE  
  PEUGEOT 106  
  CEDRIC  
  CELINE  
  LUDOVIC  
-- 3  
  STRASBOURG  
  SEVERINE
```

Remarques

Le TP libre est à déposer – au plus tard -pour le **dimanche 5 mai 2024, 23h59**. Vous déposerez sur moodle un seul fichier : le code source commentés du programme (un seul fichier .c).

Le projet doit être réalisé seul.

La note est fonction de la qualité *du travail personnel* réalisé. Chaque fonction sera évaluée par un code de test. Il faut que le programme compile ! Et il faut impérativement respecter les profils des opérations. La gestion de la mémoire est également vérifiée.