TP2 : Relations Binaires

Préliminaires.
Le TP est à effectuer en C++ à l'aide de vos outils habituels.

Dans ce TP, nous allons travailler sur les *relations binaires* d'un ensemble. Une relation binaire R sur un ensemble E est un sous ensemble de $E \times E = \{(a,b); a,b \in E\}$. Pour exprimer que i et j sont en relations par R, on peut écrire

— $(i,j) \in R$;
— $R(i,j)$;
— iRj.

Exemples :

— (ex1) $R(i,j) \Leftrightarrow i = j$;
— (ex2) $R(i,j) \Leftrightarrow i \leq j$;
— (ex3) $R(i,j) \Leftrightarrow i$ est un diviseur de j ;
— (ex4) $R(i,j) \Leftrightarrow i$ et j ont le même reste modulo 4.

Pour représenter une relation binaire en `C++`, nous utilisons un double tableau de booléen de type `vector<vector<bool> >` et représentons la relation sous forme d'une **matrice carrée** : la case `R[i][j]` est à `true` si i est en relation avec j.

Par exemple, voici une relation représentée par une matrice et par un ensemble de couples (i,j) (les couples sont les cases de la matrices égales à 1) :

$$R = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \{(0,0),(0,1),(1,0),(1,2),(2,2)\}.$$

(1) Téléchargez depuis le site l'archive `TP2.zip` et extrayez les fichiers dans votre dossier personnel.

(2) Ouvrez le fichier `TP2.cpp` et vérifiez la compilation.

(3) Remarquez la ligne

```
typedef vector<vector<bool>> Relation;
```

Elle sert à **définir** un nouveau **type** `Relation` que nous utiliserons tout au long du TP. **A chaque fois qu'il est écrit** `Relation`**, c'est comme s'il était écrit** `vector<vector<bool> >`. Pour créer une nouvelle relation, on pourra utiliser la fonction `emptyRelation` :

```
Relation R = emptyRelation(5) /* relation vide sur {0,1,2,3,4} */
```

On peut ensuite ajouter un couple (i,j) de cette façon :

```
R[i][j] = true;
```

De même, on peut tester si i est en relation avec j de cette façon :

```
if(R[i][j]) {
        ...
}
```

Pour parcourir la matrice en entier, on utilise une **double boucle** :

```
for(int i = 0; i < R.size(); i++) {
        for(int j = 0; j < R.size(); j++) {
                ...
        }
}
```

(4) De nombreux exemples ont été définis dans le fichier qui nous serviront de test pour les fonctions à écrire. **Observez les exemples pour comprendre le lien entre la matrice et la relation** : une valeur en position i, j est `true` ssi i est en relation avec j (on compte les lignes et les colonnes à partir de 0).

Exercice 1 (Affichage et exemples)**.**
Dans ce premier exercice, on se familiarise avec l'objet `Relation` et on crée nos premiers exemples.

(1) Une fonction `afficheMatriceRelation` vous est donnée en exemple, lancez le programme en exécutant le test `testAfficheRelation`.

(2) Bien que ce soit un format très pratique pour les calculs, la forme matricielle n'est pas toujours très lisibles par un humain. On se propose donc d'implanter la fonction `afficheCouplesRelations` qui donne explicitement tous les couples (i, j) tels que `R[i][j]` soit à `true`. Par exemple, sur `ex1`, la fonction doit afficher $(0, 0)(1, 1)(2, 2)$ sur `ex2`, elle doit afficher $(0, 0)(0, 1)(0, 2)(1, 1)(1, 2)(2, 2)$. Implantez la fonction et lancez la fonction de test correspondante.

(3) Le fichier fourni plusieurs exemples pour des matrices de relations. Cependant, on aura besoin de tester sur des exemples plus grand et pour cela, on décide de créer quelques fonctions. La fonction `exemplek` vous est donnée en exemple. Implantez la fonction `plusPetitEgal` qui retourne une matrice relation de taille n telle que `R[i][j]` soit à `true` ssi $i \leq j$. (Pensez à utiliser la fonction `emptyRelation` pour créer une nouvelle relation de taille donnée). Lancez les tests correspondants.

(4) Sur le même principe, implantez la fonction `diviseur` qui retourne une matrice telle que `R[i][j]` soit à `true` ssi i est un diviseur de j et lancez les tests correspondants.

Rappel : vous pouvez tester si i divise j en regardant le reste de la division euclidienne `j%i` est égal à 0. **Attention** il n'est pas possible d'effectuer des divisions ou reste par 0 : on considère que 0 n'est diviseur **d'aucun** nombre (pas même lui même).

(5) Implantez la fonction `modk` qui prend en paramètre la taille n et un entier k et retourne une relation telle que i soit en relation avec j ssi i et j ont le même reste par la division par k (`i%k == j%k`). Lancez les tests correspondants.

Exercice 2 (Premières propriétés)**.**
Nous allons implanter les tests les plus simples sur les propriétés des relations.

(1) On dit qu'une relation R est incluse dans une relation S si *l'ensemble des couples en relation pour R est inclus dans l'ensemble des couples en relation pour S*. En terme de matrice, cela signifie que si `R[i][j]` est `true` alors `S[i][j]` doit aussi être `true`, autrement dit, d'un point de vue logique $R(i, j) \Rightarrow S(i, j)$. Notez que l'implication ne va que dans un sens, on peut très bien avoir `R[i][j]` à `false` et `S[i][j]` à `true`.

Exemple :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \subset \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$
$$\{(0,0),(1,1)\} \subset \{(0,0),(1,1),(1,2)\}$$

Implantez la fonction *incluse* et lancez les tests correspondants.

(2) Une relation est dite *réflexive* si $R(i,i)$ (i est en relation avec i) pour tout i (Comment cela se voit-il sur la matrice ?). Implantez la fonction `estReflexive` et lancez les tests correspondants.

(3) Une relation est dite *symétrique* si $R(i,j) \Leftrightarrow R(j,i)$ (si i est en relation avec j, alors j est en relation avec i et vice-versa) pour tout i,j (Comment cela se voit-il sur la matrice ?). Implantez la fonction `estSymetrique` et lancez les tests correspondants.

(4) Une relation est dite *antisymétrique* si on n'a jamais $R(i,j)$ et $R(j,i)$ en même temps. Implantez la fonction `estAntiSymetrique` et lancez les tests correspondants..

(5) Au delà de nous faire manipuler les propriétés des relations, le but de l'implantation est de nous permettre **d'expérimeter** sur des propriétés mathématiques et de trouver **grâce à la machine** des exemples et contre-exemple de certaines propriétés. C'est pourquoi nous allons écrire des fonctions **d'illustration**. Nous avons écrit la fonction `afficheProprietesSimples` qui affiche une relation et ses propriétés ainsi qu'un exemple d'illustration autour des notions *reflexive* et *symétrique*. En suivant cet exemple, complétez la fonction `illustrationSymetriqueAntisymetrique` avec une relation qui n'est **ni symétrique, ni anti-symétrique**. (C'est le cas d'une des relations exemples données dans le fichier).

Exercice 3 (Premières opérations)**.**
Nous allons implanter les opérations de base des relations.

(1) Implantez la fonction `inverseRel` qui retourne la relation I, inverse de R, c'est-à-dire $I(i,j)$ ssi (non $R(i,j)$).

(2) Implantez la fonction `unionRel` qui retourne la relation union de deux relations données.

(3) Implantez la fonction `intersectionRel` qui retourne la relation intersection de deux relations données.

Exercice 4 (Composition et transitivité)**.**

(1) La *composition* C de deux relations R et V est définie de la manière suivante :

$$C(i,j) \Leftrightarrow \exists k; R(i,k) \text{ et } V(k,j).$$

Implantez la fonction `compositionRel` et lancez les tests correspondants.

(2) Une relation R est dite *transitive* si pour tout i,j,k

$$R(i,k) \text{ et } R(k,j) \Rightarrow R(i,j).$$

Autrement dit, R est transitive si la composition de R et R est incluse dans R. Implantez la fonction `estTransitive` et lancez les tests correspondants.

(3) Complétez la fonction d'illustration `illustrationUnionTransitive` : l'union de deux relations transitives n'est pas toujours transitive.

(4) On dit qu'une relation est une relation *d'équivalence* si elle est réflexive, transitive et symétrique. Implantez la fonction `estEquivalence` et lancez les tests correspondants.

(5) On dit qu'une relation est une relation *d'ordre* si elle est réflexive, transitive et anti-symétrique. Implantez la fonction `estOrdre` et lancez les tests correspondants.

Exercice 5 (Cloture ♣)**.**
Le principe de *cloture* d'une relation R par une propriété P est le suivant : trouver la plus petite relation S (selon l'ordre d'inclusion) telle que S vérifie P et R est incluse dans S. Autrement dit, on cherche à rajouter le minimum de relations possibles à R pour que R vérifie P.

(1) Que doit-on rajouter à une relation R pour la rendre réflexive ? Implantez la fonction `clotureReflexive` et lancez les tests correspondants.

(2) Que doit-on rajouter à une relation R pour la rendre symétrique ? Implantez la fonction `clotureSymetrique` et lancez les tests correspondants.

(3) Comment calculer la cloture transitive d'une relation ? Implantez la fonction `clotureTransitive` et lancez les tests correspondants.

(4) Existe-il une cloture anti-symétrique ? Pourquoi ?

(5) Implantez les fonctions `clotureTransRef` et `clotureEquivalence` et lancez les tests correspondants.

Exercice 6 (Aller plus loin ♣)**.**

(1) Testez par un programme sur les exemples la compatibilité des différentes propriétés (réflexive, symétrique, anti-symétrique, transitive) avec les opérations (inverse, union, intersection, composition, cloture) et écrivez des fonctions d'illustration adéquate.

(2) Écrivez une fonction main qui permette à l'utilisateur d'entrer sa propre relation et qui lui en donne les propriétés.

Exercice 7 (Aller encore plus loin ♣♣)**.**

(1) Trouvez comment tirer un nombre aléatoire en `C++` et écrivez une fonction qui vous retourne une **relation alétoire** de taille donnée.

(2) Écrivez un programme qui effectue de **nombreux tirages aléatoires** et compte le nombre de relations reflexives, symétriques, antisymétriques, transitives pour en évaluer la fréquence.