Informatique - MPI

Exercice A1

Soit un système temps réel à n processus asynchrones $i \in [\![1, n]\!]$ et m ressources r_j. Quand un processus i est actif, il bloque un certain nombre de ressources listées dans un ensemble P_i et une ressource ne peut être utilisée que par un seul processus. On cherche à activer simultanément le plus de processus possible.

Le problème de décision `ACTIVATION` correspondant ajoute un entier k aux entrées et cherche à répondre à la question : *Est-il possible d'activer au moins k processus en même temps ?*

Question 1. Soit $n = 4$ et $m = 5$. On suppose que $P_1 = \{r_1, r_2\}$, $P_2 = \{r_1, r_3\}$, $P_3 = \{r_2, r_4, r_5\}$ et $P_4 = \{r_1, r_2, r_4\}$. Est-il possible d'activer 2 processus en même temps ? Même question avec 3 processus.

Question 2. Dans le cas où chaque processus n'utilise qu'une seule ressource, proposer un algorithme résolvant le problème `ACTIVATION`. Évaluer sa complexité.

Question 3. On souhaite montrer que `ACTIVATION` est NP-complet. Donner un certificat pour ce problème.

Question 4. Écrire en pseudo-code un algorithme de vérification polynomial. On suppose disposer de trois primitives, de complexité polynomiale chacune.

- `appartient(c,i)` renvoie `vrai` si le processus `i` est dans l'ensemble d'entiers `c`.
- `intersecte(P_i,R)` renvoie `vrai` si le processus `i` utilise une ressource d'un ensemble de ressources `R`.
- `ajoute(P_i,R)` ajoute les ressources `P_i` dans l'ensemble `R` et renvoie ce nouvel ensemble.

Question 5. En théorie des graphes, le problème `STABLE` pose la question de l'existence, dans un graphe non orienté $G = (S, A)$, d'un ensemble d'au moins k sommets ne contenant aucune paire de sommets voisins. En d'autres termes, existe-t-il $S' \subset S$, $|S'| \geq k$ tel que $s, p \in S' \Rightarrow (s, p) \notin A$? En utilisant le fait que `STABLE` est NP-complet, montrer par réduction que le problème `ACTIVATION` est également NP-complet.

Exercice A2

Le programme OCaml suivant définit n fils d'exécution qui incrémentent tous un même compteur partagé.

```ocaml
(* Nombre de fils d'exécution *)
let n = 100

(* Un même compteur partagé *)
let compteur = ref 0

(* Chaque fil d'exécution de numéro i va incrémenter le compteur *)
let f i = compteur := !compteur + 1

(* Création de n fils exécutant f associant à chaque fil son numéro *)
let threads = Array.init n (fun i -> Thread.create f i)

(* Attente de la fin de n fils d'exécution *)
let () = Array.iter (fun t -> Thread.join t) threads
```

On rappelle que l'on dispose en OCaml des trois fonctions `Mutex.create : unit -> Mutex.t` pour la création d'un verrou, `Mutex.lock : Mutex.t -> unit` pour le verrouillage et `Mutex.unlock : Mutex.t -> unit` pour le déverrouillage, du module `Mutex` pour manipuler des verrous.

Question 1. Quelles sont les valeurs possibles que peut prendre le compteur à la fin du programme.

Question 2. Identifier la section critique et indiquer comment et à quel endroit ajouter des verrous pour garantir que la valeur du compteur à la fin du programme soit n de manière certaine.

Dans la suite de l'exercice, on suppose que l'on ne dispose pas d'une implémentation des verrous. On se limite au cas de deux fils d'exécution, numérotés 0 et 1. Nous cherchons à garantir deux propriétés :

- *Exclusion mutuelle* : un seul fil d'exécution à la fois peut se trouver dans la section critique ;
- *Absence de famine* : tout fil d'excéution qui cherche à entrer dans la section critique pourra le faire à un moment.

On utilise pour cela un tableau `veut_entrer` qui indique pour chaque fil d'exécution s'il souhaite entrer en section critique ainsi qu'une variable `tour` qui indique quel fil d'exécution peut effectivement entrer dans la section critique. On propose ci-dessous deux versions modifiées `f_a` et `f_b` de la fonction `f`, l'objectif étant de pouvoir exécuter `f_a 0` et `f_a 1` de manière concurrente, et de même pour `f_b`.

```
let veut_entrer = [|false; false|]
let tour = ref 0

let f_a i =
  let autre  = 1 - i in
  veut_enter.(i) <- true;
  while veut_entrer.(autre) do () done;
  (* section critique *)
  veut_entrer.(i) <- false

let f_b i =
  let autre = 1 - i in
  veut_entrer.(i) <- true;
  tour := i;
  while veut_entrer.(autre) && !tour = autre do () done;
  (* section critique *)
  veut_entrer.(i) <- false
```

Question 3. Expliquer pourquoi aucune de ces deux versions ne convient, en indiquant la propriété qui est violée.

Question 4. Proposer une version `f_c` qui garantit les deux propriétés.

Question 5. Rappeler le principe d'un algorithme permettant de généraliser à n fils d'exécution.

Exercice A3

On considère le schéma de base de données suivant, qui décrit un ensemble de fabricants de matériel informatique, les matériels qu'ils vendent, leurs clients et ce qu'achètent leurs clients. Les attributs des clés primaires des six premières relations sont soulignés.

```
Production(NomFabricant, Modele)
Ordinateur(Modele, Frequence, Ram, Dd, Prix)
Portable(Modele, Frequence, Ram, Dd, Ecran, Prix)
Imprimante(Modele, Couleur, Type, Prix)
Fabricant(Nom, Adresse, NomPatron)
Client(Num, Nom, Prenom)
Achat(NumClient, NomFabricant, Modele, Quantite)
```

Chaque client possède un numéro unique connu de tous les fabricants. La relation `Production` donne pour chaque fabricant l'ensemble des modèles fabriqués par ce fabricant. Deux fabricants différents peuvent proposer le même matériel. La relation `Ordinateur` donne pour chaque modèle d'ordinateur la vitesse du processeur (en Hz), les tailles de la Ram et du disque dur (en Go) et le prix de l'ordinateur (en €). La relation `Portable`, en plus des attributs précendents, comporte la taille de l'écran (en pouces). La relation `Imprimante` indique pour chaque modèle d'imprimante si elle imprime en couleur (vrai/faux), le tye d'impression (laser ou jet d'encre) et le prix (en €). La relation `Fabricant` stocke les nom et adresse de chaque fabricant, ainsi que le nom de son patron. La relation `Client` stocke les noms et prénoms de tous les clients de tous les fabricants. Enfin, la relation `Achat` regroupe les quadruplets (client c, fabriquant f, modèle m, quantité q) tels que le client de numéro c a acheté q fois le modèle m au fabricant f. On suppose que l'attribut `Quantite` est toujours strictement positif.

Question 1. Proposer une clé primaire pour la relation `Achat` et indiquer ses conséquences en terme de modélisation.

Question 2. Identifier l'ensemble des clés étrangères éventuelles de chaque table.

Question 3. Donner en SQL des requêtes répondant aux question suivantes :
□ **3.1.** Quels sont les numéros des modèles des matériels (ordinateur, portable ou imprimante) fabriqués par l'entreprise du nom de Durand ?
□ **3.2.** Quels sont les noms et adresses des fabricants produisant des portables dont le disque dur a une capacité d'au moins 500 Go ?
□ **3.3.** Quels sont les noms des fabricants qui produisent au moins 10 modèles différents d'imprimantes ?
□ **3.4.** Quels sont les numéros des clients n'ayant acheté aucune imprimante ?

Exercice B1

Cet exercice est à traiter dans le langage C. Le code de certaines fonctions décrites dans l'énoncé est donné dans un fichier joint `laby.c`. *Il est à compléter avec les fonctions demandées.*

Question 1. On implémente une structure *union-find* naïve pour gérer une partition de $[\![0, n-1]\!]$ à l'aide d'un tableau de taille n contenant en case i la classe de l'élément i, définie comme étant le plus petit numéro qui est dans la même classe que i. Par exemple le tableau $[0, 0, 2, 0]$ représente la partition $\{\{0, 1, 3\}, \{2\}\}$ de $[\![0, 3]\!]$. Le code définit une telle structure `uf`.

□ **1.1.** Écrire une fonction `uf* initialiser_partition(int n` créant une partition de $[\![0, n-1]\!]$ dans laquelle chaque élément est seul dans sa classe puis une fonction `void liberer_partition(uf* p)` qui libère toute la mémoire que cette fonction d'initialisation a alloué sur le tas pour un objet de type `uf*`.

□ **1.2.** Écrire une fonction `int trouver_classe(uf* partition, int elem)` renvoyant la classe d'un élément.

□ **1.3.** Écrire une fonction `void fusionner_classes(uf* partition, int i, int j) partition` qui fusionne les classes des éléments `i` et `j` en modifiant `partition`. On la testera sur un exemple pertinent.

□ **1.4.** Quelles sont les complexités des fonctions d'initialisation, de recherche et d'union dans ce contexte ?

Question 2. On cherche à présent à construire des labyrinthes sur des grilles carrées dans lesquels il est garanti qu'il existe un unique chemin simple entre le carré en haut à gauche et le carré en bas à droite (on ne se déplace pas en diagonale). La figure 1 présent un labyrinthe de côté 4, les traits en gras représentant les murs, et sa représentation par un graphe. Ainsi, un labyrinthe de côté n est encodé par un graphe non orienté à n^2 sommets correspondant à une numérotation de haut en bas et de gauche à droite des cases de la grille. On peut passer de la case u à la case v dans le labyrinthe si et seulement si il y a une arête entre u et v dans le graphe correspondant. Les murs externes ne sont pas encodés : il y en a par défaut partout sauf en haut à gauche et en bas à droite. La structure `graphe` utilisée pour représenter le graphe encodant un labyrinthe est donnée dans le code. Le code fournit également une fonction de création d'un (pointeur sur un) labyrinthe (de type `graphe*`) ne contenant aucune arête (donc avec des murs partout) et de libération d'un tel objet.

□ **2.1.** Écrire une fonction `void coordonnees(int s, int n, int* i, int* j)` stockant les coordonnées (i, j) du sommet s dans i et j respectivement en sachant que s est le numéro d'une case dans une grille de côté `n`. Par exemple, les coordonnées du sommet 13 dans une grille de côté 4 sont $(3, 1)$.

□ **2.2.** La fonction `sont_cote_a_cote` fournie par le code compagnon renvoie un booléen indiquant si deux sommets sont côte à côte dans une grille donnée. Par exemple, dans une grille de côté 4, 8 et 4 sont côte à côte mais pas 3 et 4. Déterminer la complexité temporelle pire cas de `sont_cote_a_cote`.

Question 3. On propose à présent de créer un labyrinthe de côté n aléatoirement à l'aide de la procédure suivante.

- Initialiser un labyrinthe L de côté n (donc à n^2 sommets) avec des murs partout.
- Initialiser une partition P des sommets.
- Pour chacune des n^4 arêtes possibles (u, v), prises dans un ordre aléatoire.
- Si u et v sont côte à côte et que u et v ne sont pas dans la même classe d'après la partition P.
- Casser le mur entre u et v dans L.
- Renvoyer L.

□ **3.1.** Compléter la fonction `graphe* creer_laby(int n)` dans le code compagnon en suivant cette stratégie. Le code propose une implémentation d'une fonction `melanger` qui permet le tirage aléatoire d'arêtes.

□ **3.2.** Créer un labyrinthe de côté 3. Le dessiner. On pourra utiliser la fonction `afficher_matrice`.

□ **3.3.** Quelle est la complexité de `creer_laby` ?

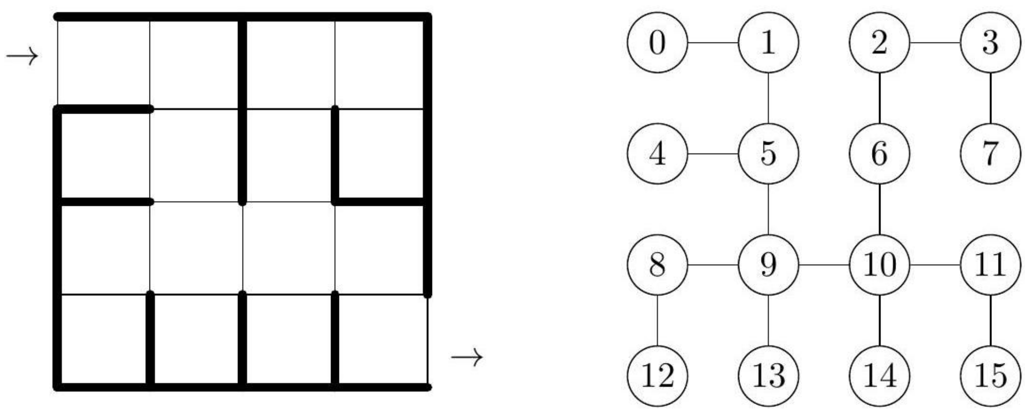

Figure 1 – Labyrinthe exemple.

Exercice B2

Cet énoncé est accompagné d'un code compagnon en OCaml `localite.ml` fournissant le type décrit par l'énoncé et quelques fonctions auxiliaires. Il est à compléter en y implémentant les fonctions demandées. On privilégiera un style de programmation fonctionnel.

On considère un alphabet Σ. Si L est un langage sur Σ, on note :

- $P(L) = \{a \in \Sigma \mid a\Sigma^\star \cap L \neq \varnothing\}$
- $D(L) = \{a \in \Sigma \mid \Sigma^\star a \cap L \neq \varnothing\}$
- $F(L) = \{m \in \Sigma^2 \mid \Sigma^\star m \Sigma^\star \cap L \neq \varnothing\}$
- $N(L) = \Sigma^2 \setminus F(L)$

On rappelle qu'un langage L est dit *local* si et seulement si l'égalité de langages suivantes est vérifiée :

$$L \setminus \{\varepsilon\} = (P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L) \Sigma^\star)$$

Question 1. Calculer les ensembles $P(L)$, $D(L)$, $F(L)$ et $N(L)$ dans le cas où L est le langage dénoté par l'expression régulière $a^\star(ab)^\star \mid aa$ sur l'alphabet $\{a, b\}$. Ce langage est-il local ? On vérifiera la cohérence entre les réponses à cette question et celles obtenues via les fonctions demandées dans la suite de l'énoncé.

Dans la suite de l'exercice, on cherche à concevoir un algorithme répondant à la spécification suivante.

$\begin{cases} \textbf{Entrée :} \text{ Une expression régulière } e \text{ sur un alphabet } \Sigma \text{ ne faisant pas intervenir le symbole } \varnothing. \\ \textbf{Sortie :} \text{ Vrai si le langage dénoté par } e \text{ est local, faux sinon.} \end{cases}$

Par défaut, dans la suite de l'énoncé, *expression régulière* signifie *expression régulière ne faisant pas intervenir le symbole* $\varnothing$. Les expressions régulières seront manipulées en OCaml via le type somme suivant.

```
type regexp =
  | Epsilon
  | Letter of string (*La chaîne en question sera toujours de longueur 1*)
  | Union of regexp * regexp
  | Concat of regexp * regexp
  | Star of regexp
```

On propose tout d'abord de calculer les ensembles $P(L)$, $D(L)$ et $F(L)$ à partir d'une expression régulière dénotant L. Ces ensembles sont représentés par des listes de chaînes de caractères qui vérifient les propriétés suivantes : elles sont triées dans l'ordre croissant selon l'ordre lexicographique ; elles sont sans doublons.
L'énoncé fournit une fonction `union` permettant de calculer l'union sans doublons de deux listes triées. La définition inductive d'une expression régulière invite à calculer inductivement les ensembles $P(L)$, $D(L)$ et $F(L)$. C'est ce que propose la fonction `compute_P` fournie par l'énoncé.

Question 2. En supposant que la fonction `contains_epsilon : regexp -> bool` renvoie `true` si et seulement si le langage dénoté par l'expression régulière en entrée contient le mot ε, justifier brièvement la correction de `compute_P`.

Question 3. Implémenter la fonction `contains_epsilon`.

Question 4. Sur le modèle de `compute_P`, implémenter une fonction `compute_D : regexp -> string list` permettant le calcul de l'ensemble $D(L)$ étant donnée une expression régulière dénotant le langage L.

Question 5. Expliquer en langage naturel comment calculer récursivement l'ensemble $F(L)$ étant donnée une expression régulière e dénotant le langage L. Si $e = e_1 e_2$ on pourra exprimer $F(L)$ en fonction notamment de $P(L_2)$ et $D(L_1)$ où L_1 (resp. L_2) est le langage dénoté par e_1 (resp. e_2).

Question 6. Écrire une fonction `prod : string list -> string list -> string list` calculant le produit cartésien des deux listes en entrée, qu'on pourra supposer triées dans l'ordre lexicographique croissant et sans doublons, puis qui pour chaque couple de chaînes dans la liste obtenue les concatène. Par exemple :

```
prod ["a";"c";"e"] ["b";"c"] = ["ab";"ac";"cb";"cc";"eb";"ec"]
```

Question 7. En déduire une fonction `compute_F : regexp -> string list` déterminant l'ensemble $F(L)$ étant donnée une expression régulière dénontant le langage L.

Dans les questions qui suivent, on ne demande PAS d'implémenter les algorithmes décrits.

Question 8. Décrire en pseudo-code ou en langage naturel un algorithme permettant de calculer un automate reconnaissant $(P(L)\Sigma^\star \cap \Sigma^\star D(L)) \setminus (\Sigma^\star N(L) \Sigma^\star)$ étant donnée une expression régulière dénotant L.

Question 9. Décrire un algorithme qui détecte si le langage dénoté par une expression régulière est local ou non.

Question 10. Pourquoi est-il légitime de ne considérer que les expressions régulières ne faisant pas intervenir $\varnothing$? Comment modifier l'algorithme obtenu dans le cas où cette contrainte n'est plus vérifiée ?