INF431 : Composition d'informatique 1

Promotion 2007

Sujet proposé par Laurent Mauborgne

29 avril 2009

Les sections 1, 2 et 3 sont largement indépendantes et peuvent être traitées dans l'ordre de votre choix.

On dispose d'une base de données qui décrit un réseau ferré et les trains qui circulent sur ce réseau. La base de données associe en particulier à chaque train son nom et son trajet détaillé sous forme de liste triée de couples gare horaire.

1 Vérification des horaires

Pour les expressions régulières demandées dans cette section, on utilisera le symbole + pour l'alternative et la simple juxtaposition pour la concaténation.

On souhaite vérifier la cohérence entre la base de données et les tableaux d'horaires affichés ou distribués dans les stations. Pour cela, on choisit de scanner ces tableaux et d'utiliser un logiciel de reconnaissance de caractères. On obtient donc une suite de lignes de caractères, certaines intéressantes pour la vérification des horaires, d'autres sans intérêt.

LES ARCS-DRA		**06.23**	**06.32**		**06.49**	***07.21***	**07.32**	***07.49***	**08.45**	**09.05**	**09.52**
Fréjus		06.37			07.04		07.47		09.00	09.20	10.08
ST-RAPHAËL-V	**06.05**	**06.42**	***06.51***	**07.10**	**07.10**	***07.39***	**07.52**	***08.10***	**09.06**	**09.25**	**10.14**
Boulouris/Mer	06.09			07.14	07.14		07.56			09.29	
Le Dramont							08.01			09.33	
AGAY				**07.19**	**07.19**		**08.04**			**09.37**	
ANTHÉOR-CA							**08.09**			**09.41**	
Le Trayas							08.14			09.47	
Théoule-sur-Mer							08.20			09.52	
Mandelieu-la-N	06.25	07.00		07.31	07.31		08.23			09.55	
Cannes-La-Bocc	06.30			07.36	07.36		08.28			10.00	
CANNES	**06.36**	**07.09**	***07.20***	**07.51**	**07.51**	***08.05***	**08.33**	***08.40***	**09.30**	**10.07**	**10.39**
Golfe-Juan-Vall	06.42			07.58	07.58		08.39			10.14	
Juan-les Pins	06.46			08.02	08.02		08.43			10.18	
ANTIBES	**06.49**	**07.20**	***07.30***	**08.05**	**08.05**	***08.18***	**08.47**	***08.53***	**09.39**	**10.21**	**10.49**
Biot	06.53			08.09	08.09					10.25	
Villeneuve-Loub	06.57			08.12	08.12					10.29	
CAGNES/MER	**07.01**	**07.29**		**08.17**	**08.17**		**08.54**			**10.33**	**10.57**
Cros-de-Cagnes	07.03			08.20	08.20					10.36	
St-Laurent-du-Va	07.07			08.23	08.23		08.58			10.40	
Nice-St-Augustin	07.10			08.27	08.27		09.02			10.43	
NICE-VILLE (A	**07.16**	**07.41**	***07.45***	**08.32**	**08.32**	***08.37***	**09.07**	***09.12***	**09.52**	**10.49**	**11.06**

FIG. 1 – Exemple de tableau d'horaires.

Un exemple typique de la partie intéressante est donné en figure 1. La première ligne sera alors reconnue en "`| LES ARCS-DRA| |06.23|06.32| ... |`". Dans les tableaux rencontrés, les horaires peuvent être affichés avec ou sans 0 initial et en séparant les heures des minutes par un point ou par un h.

On donne une expression régulière pour les chiffres :

$$C = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$$

Question 1.

a) Écrivez une expression régulière permettant de reconnaître un horaire décrit comme une heure codée par un ou deux chiffres, avec ou sans 0 initial, un séparateur pouvant être un point ou un h, puis des minutes codées par deux chiffres.

b) Peut-on exprimer la contrainte qu'une minute est toujours comprise entre 0 et 59 ? Si oui, donner l'expression régulière correspondante. Et pour les heures comprises entre 0 et 23 ? Donnez l'expression la plus précise possible pour les horaires.

Corrigé.

a) H = (CC+C)(.+h)CC

b) H = (C+1C+0C+2(0+1+2+3))(.+h)(0+1+2+3+4+5)C

Après la reconnaissance de caractères, une ligne intéressante va donc comprendre un nom de station, des blancs, des horaires, et entre ces composants des caractères `'|'`. Formellement, le format d'une ligne respecte toujours les règles suivantes :

- une ligne commence et finit toujours par |,
- une ligne contient toujours au moins une case horaire, éventuellement vide,
- on peut trouver des blancs avant et après le nom de la station,
- un mot est une suite (non vide) de tous caractères autres que blanc et |,
- un nom de gare peut être composé de plusieurs mots, séparés par un blanc,
- il peut y avoir des blancs avant ou après un horaire et dans une case vide.

Question 2. En supposant donnée l'expression régulière pour les mots, que l'on écrira M, écrivez une expression régulière pour les lignes intéressantes des tableaux d'horaire. Par commodité, on écrira # pour les caractères blancs et H pour l'expression régulière définie à la question précédente.

Corrigé. B = #*

L = |BM(BM)*B|B(|B+|BHB)*|

Si on a bien identifié les lignes intéressantes, on peut vérifier que les horaires indiqués sont corrects, c'est-à-dire qu'ils correspondent bien à des trains de la base de données qui s'arrêtent aux stations à l'horaire indiqué (on n'essaiera pas de vérifier la cohérence sur les trains et on ne s'occupe pas des cases vides). Pour faire cette vérification, un tableau de trains n'est pas la structure de données la plus adaptée.

Question 3. Quelle structure de données serait adaptée à la vérification des horaires ? Décrivez-la avec précision et justifiez en quelques lignes par des arguments de complexité.

Corrigé. Le problème consiste à vérifier si un couple station horaire est bien dans l'ensemble des couples possibles. Il suffit donc de trouver une structure de données pour représenter un ensemble et pour lequel le test d'appartenance est efficace. On pourra choisir une table de hachage de ces couples, pour laquelle le test d'appartenance est de complexité constante.

2 Trier les trains

Dans cette section, on demande d'écrire les réponses en Java. On supposera que le paquetage dans lequel seront écrites vos réponses contient déjà une classe `Train`*, avec les méthodes* `hashCode()` *et* `equals(Object o)` *redéfinies pour correspondre à l'égalité des trains.*

On pourra utiliser les classes de librairie :

- `Integer`*. On rappelle que si on a un* `Integer i`*, on peut affecter* `i=0` *et calculer* `i = i+1`*.*
- `LinkedList<E>`*. On rappelle les méthodes* `add(E o)` *qui ajoute un élément en queue de liste,* `E remove()` *qui renvoie l'élément de tête de la liste et le supprime de la liste, et* `boolean isEmpty()` *qui retourne* `true` *si la liste est vide. Ces méthodes ont un coût constant.*
- `HashMap<K,V>`*, avec les méthodes* `V get(Object key)`, `put(K key, V value)` *et* `Set<K> keySet()`*. Si* `key` *n'est pas dans la table* `t`*, alors* `t.get(key)` *retourne* `null`*. Si* `key` *est déjà dans la table* `t`*, alors* `t.put(key,value)` *remplace l'ancienne valeur associée à* `key` *par* `value`*. On pourra considérer que ces trois méthodes ont un coût constant.*

On cherche maintenant à afficher les trains triés. On ne sait pas à l'avance quel ordre sera employé pour le tri, mais on souhaite tout de même écrire une procédure utilisable quel que soit cet ordre. Pour être le plus général possible, on souhaite que ces ordres puissent être partiellement décrits par une relation "après" qui ne soit pas forcément transitive. Cela signifie que si la relation B "après" A est vraie alors il faudra forcément afficher A avant B dans le tri, mais si cette relation est fausse, on n'a aucune contrainte.

Par exemple, on souhaite afficher un train A avant un train B dès qu'on peut, prenant le trains A et en suivant des correspondances (un nombre quelconque de correspondances), prendre le train B. Le calcul exact de l'ordre serait très coûteux, on se contente donc de spécifier une relation locale qui dit que B "après" A si il existe une station où A et B s'arrêtent et l'horaire de A est plus tôt que l'horaire de B pour cette station.

On choisira d'encoder cette relation par une fonction qui prend en argument un train et retourne la liste des trains qui viennent après lui. La relation "après" est représentée naturellement par un graphe sur l'ensemble des trains, qu'on peut ensuite exploiter pour trier efficacement les trains, mais les appels à cette fonction peuvent être coûteux. On souhaite donc encoder une fois pour toutes ce graphe sous une forme plus efficace.

On se placera dans un cadre où le nombre de trains sera très grand. En particulier on considérera qu'il est trop coûteux de représenter l'ensemble de tous les couples de trains par un tableau à deux dimensions codant une relation entre tous les trains.

Question 4. Proposez une classe `GrapheTrains` pour représenter un graphe dont les sommets sont des `Train`. On ne demande pas de constructeur ni de méthode pour l'instant.

Corrigé.

```
class GrapheTrains {
  HashMap<Train, LinkedList<Train>> suivants;
}
```

Pour écrire un tri général, le constructeur du graphe doit prendre en argument une méthode qui définit la relation utilisée. On choisira d'utiliser une méthode "après" qui prend en argument un train et retourne la liste des trains qui viennent après lui. En Java, une relation sera donc représentée par un objet d'une classe qui implante une interface qui déclare cette méthode.

Question 5. Écrivez une interface `Relation` pour les relations sur les `Train` et le constructeur de `GrapheTrains` qui prend en argument l'ensemble des trains connus et une relation sur ces trains. Pour l'ensemble des trains connus, on pourra choisir entre un tableau ou un `Set`.

Corrigé.

```
interface Relation {
  LinkedList<Train> apres(Train t);
}

class GrapheTrains {
  HashMap<Train, LinkedList<Train>> suivants;

  GrapheTrains(Set<Train> trains, Relation r) {
    for(Train t:trains)
      suivants.put(t, r.apres(t));
  }

  GrapheTrains(Train[] trains, Relation r) {
    for(int i=0; i<trains.length; i++) {
      suivants.put(trains[i], r.apres(trains[i]));
  }
}
```

La relation ne peut servir à trier que si elle ne contient pas de cycle.

Question 6. Écrivez une fonction dans la classe `GrapheTrains` qui indique si le graphe contient un cycle. Cette fonction renvoie `true` si le graphe contient un cycle et `false` sinon. Elle devra avoir un coût linéaire en le nombre d'arêtes du graphe.

Corrigé. Question de cours : il suffit de faire un parcours et de voir si on retourne sur ses pas.

```
final static Integer ENCOURS = 0, FINI = 1;

boolean parcours(HashMap<Train, Integer> visite, Train t) {
  Integer etat = visite.get(t);
  if(etat == ENCOURS) return true;
  if(etat == FINI) return false;
  visite.put(t, ENCOURS);
  for(Train u:suivants.get(t)) {
    if(parcours(visite, u)) return true;
  }
  visite.put(t, FINI);
  return false;
}

boolean contientCycle() {
  HashMap<Train, Integer> visite = new HashMap<Train, Integer>();
  for(Train t:suivants.keySet()) {
    if(visite.get(t) == null) {
      if(parcours(visite, t)) return true;
    }
  }
  return false;
}
```

D'une façon naturelle, tout tri des trains commence par un train qui n'en a aucun avant lui. On va donc calculer pour chaque train le nombre de trains directement avant lui. On appellera ce nombre le degré entrant du sommet dans le graphe.

Question 7. Écrivez une méthode `calculDegres()` qui calcule le degré de chaque sommet, le stocke dans une structure appropriée et retourne cette structure.

Corrigé. La structure choisie pour les degrés sera `HashMap<Train, Integer> degre`.

```
HashMap<Train, Integer> calculDegres() {
  Hashmap<Train, Integer> degre = new HasMap<Train, Integer>();
  for(Train t:suivants.keySet()) degre.put(t, 0);
  for(Train t:suivants.keySet()) {
    for(Train u:suivants.get(t)) degre.put(u, degre.get(u)+1);
  }
  return degre;
}
```

Le tri va donc consister à choisir un sommet S de degré nul, l'afficher, puis à faire le tri du graphe dans lequel on aurait supprimé S. Comme seul le degré compte pour ce tri, on ne supprime pas vraiment S, mais on ajuste les degrés des autres sommets pour qu'ils ignorent les arcs venant de S.

Question 8. Écrivez le tri du graphe dans la classe `GrapheTrains` en suivant cet algorithme. Votre méthode peut se contenter d'afficher les trains en suivant cet ordre. Calculez la complexité de votre fonction. (Vous aurez plus de points si votre fonction est linéaire.)

Corrigé. La seule difficulté consiste à trouver facilement les sommets de degré nul. Pour cela, un simple liste contenant ces sommets suffit.

```
void tri() {
  LinkedList<Train> degresNuls = new LinkedList<Train>();
  HashMap<Train, Integer> degre = calculDegres();
  for(Train t:suivants.keySet()) {
     if(degre.get(t)==0) degresNuls.add(t);
  }
  while(!degresNuls.isEmpty()) {
    Train t = degresNuls.remove();
    System.out.println(t);
    for(Train u:suivants.get(t)) {
      Integer d = degre.get(u) - 1;
      degre.put(u, d);
      if(d==0) degresNuls.add(u);
    }
  }
}
```

La première boucle est de complexité le nombre de sommets. La seconde boucle est exécutée au plus autant de fois que le nombre de sommets, car à chaque tour un sommet est retiré et un sommet retiré ne peut être rajouté à nouveau (car on ne touche plus à son degré). De plus, pour chaque sommet, la boucle interne est exécutée autant de fois que le nombre de voisins. Cela donne une borne de l'ordre du nombre d'arcs du graphe.

3 Rationalisation du réseau

Dans cette section, quand on demande des fonctions ou des procédures, on attend une réponse en pseudo-code.

On dispose maintenant, en plus des trains, d'une description du réseau ferroviaire sous forme de graphe dont les sommets sont des stations et dont chaque arête représente une voie ferrée entre deux stations. On appellera $\mathcal{R}$ ce graphe, et $\mathcal{R}.A$ l'ensemble de ses arêtes. On suppose que les voies peuvent être empruntées dans les deux sens. Dans toutes les questions, on supposera toujours que ce graphe est connexe.

Question 9. Écrivez une fonction qui prend en argument le réseau et l'ensemble des trains et retourne l'ensemble des voies qui ne sont utilisées par aucun train.

Corrigé.

fonction INUTILES(R, T)
$res \leftarrow R.A$
pour chaque $t \in T$
faire
- $g \leftarrow$ station de t
- $t \leftarrow$ suite de t
- **tant que** $t \neq \emptyset$
  - **faire**
    - $g' \leftarrow$ station de t
    - $t \leftarrow$ suivant de t
    - enlever (g, g') de res
    - $g \leftarrow g'$

renvoyer res

Pour l'ensemble d'arcs, il faut pouvoir enlever facilement des éléments. On peut pour cela utiliser un marquage des arcs par exemple.

On cherche à rationaliser le réseau en supprimant des voies inutiles. Toutefois, on souhaite garder une certaine redondance : il faut pouvoir relier deux stations même si une voie ou une troisième station est bloquée.

Question 10. Montrez que si le réseau contient au moins 3 stations et si l'on peut toujours relier deux stations quand une autre station est bloquée, alors on peut toujours le faire quand une voie est bloquée.

Corrigé. En effet, bloquer une station revient à bloquer toutes les voies ayant cette station comme extrémité. Donc si pour tout couple de stations on connaît un chemin qui évite une station donnée, on peut aussi éviter une voie donnée en évitant une des stations extrémité de cette voie. Reste le cas où les deux stations sont à l'extrémité de cette voie. On va noter g et f ces stations. On sait qu'il en existe une troisième, h· On peut aller de g à h sans passer par f, puis de h à f sans passer par g. On peut donc aller de f à g sans passer par la voie reliant f et g.

On appellera *station indispensable* une station g telle que si on enlève g du réseau, deux autres stations ne sont plus connectées. Dans un premier temps, on va les identifier à l'aide d'un parcours en profondeur du graphe.

Question 11. Montrez qu'une station g est indispensable si et seulement si pour tout parcours qui numérote les sommets dans un ordre préfixe $gg_1g_2 \dots g_n$, il existe $i > 1$ tel que pour tout $k < i$, il n'existe pas de voie reliant g_k et g_i.

Corrigé. Si il existe un tel i, alors tout chemin de g_1 à g_i passe forcément par g, donc g est indispensable. Si un tel i n'existe pas, alors par induction il existe toujours un chemin de g_1 à g_j sans passer par g pour tout $j > 1$. Si on prend maintenant deux stations distinctes de g, alors on peut aller de la première à g_1 sans passer par g, puis de g_1 à la deuxième toujours sans passer par g, on peut donc toujours relier ces deux stations même si g est bloqué.

Question 12. Écrivez une fonction qui prend en argument un réseau connexe et une station du réseau et décide si cette station est indispensable. Justifiez en quelques mots la correction de votre algorithme.

Corrigé. Suivant la question précédente, on fait une dfs partant de la station que l'on cherche à tester. Il suffit en fait de faire une dfs partant d'un voisin de cette station, en marquant comme visitée la station à tester. Si cette dfs visite tout le réseau, alors la station n'est pas indispensable, sinon elle l'est.

fonction DFS(V, g)
 si $g \in V$ **alors renvoyer** V
 $V \leftarrow V \cup \{g\}$
 pour chaque $f \in$ voisins(g)
  faire $V \leftarrow$ DFS(V, f)
 renvoyer V
fonction StationIndipensable(g)
 $f \leftarrow$ un voisin de g
 $V \leftarrow$ DFS($\{g\}, f$)
 pour chaque $f \in$ voisins de g
  faire si $f \notin V$ **alors renvoyer vrai**
 renvoyer faux

Cette caractérisation n'est pas suffisante pour déterminer si le réseau auquel on a enlevé une voie contient ou non une station indispensable en temps linéaire. Pour identifier efficacement les stations indispensables, on va s'appuyer sur l'arbre couvrant associé à un parcours en profondeur du graphe et sur la notion de point d'attache. Pour un parcours donné, le point d'attache d'un sommet g est le sommet f de plus petit numéro (pour le parcours), tel qu'on peut aller de g à f en suivant les arêtes du graphe et tel que sur le chemin on ne diminue le numéro de parcours qu'une fois.

Question 13. Montrez que le point d'attache d'un sommet g est le sommet de plus petit numéro dans le parcours parmi d'une part les voisins de g et d'autre part les points d'attache des voisins de g de plus grand numéro que g.

Corrigé. Soit f le point d'attache de g. Alors, soit f est un voisin de g, soit le chemin de g à f passe par un sommet h voisin de g. Mais ce sommet h ne peut pas être de numéro plus petit que g, car alors sur le chemin on ne peut plus qu'augmenter ce numéro et donc

h serait le point d'attache. f est donc le sommet de plus petit numéro tel qu'on peut aller de h à f en diminuant au plus une fois ce numéro. C'est donc le point d'attache de h. f est donc soit un voisin de g, soit le point d'attache d'un de ses voisins de plus grand numéro. Par minimalité du numéro du point d'attache, c'est celui de ces sommets de plus petit numéro.

Vous admettrez que les stations indispensables qui ne sont pas au départ du parcours en profondeur sont exactement celles qui sont point d'attache d'un de leurs fils dans l'arbre couvrant.

Question 14. Écrivez un algorithme qui calcule les stations indispensables d'un réseau connexe avec un coût linéaire en nombre d'arêtes du graphe. Donnez une condition naturelle pour que l'algorithme soit aussi linéaire en nombre de sommets.

Corrigé. C'est une question de cours dès qu'on a compris qu'une station indispensable est un point d'articulation, la solution est dans le poly. Pour la question de la complexité, il suffit de borner le nombre de voies partant de chaque station, ainsi on a une borne linéaire du nombre de voies par rapport au nombre de stations.

fonction DFS$(V, n, g, pere)$
 si $g \in V$ **renvoyer** $(V, n, \emptyset)$
 numero$(g) \leftarrow n$
 $n \leftarrow n + 1$
 $V \leftarrow V \cup \{g\}$
 attache$(g) \leftarrow$ numero$(pere)$
 $r \leftarrow \emptyset$ **pour chaque** f voisin de g
 faire { **si** $f \in V$ **alors** attache$(g) \leftarrow \min($attache$(g),$ numero$(f))$
 sinon { $(V, n, r') \leftarrow$ DFS(V, n, f, g)
 $r \leftarrow r \cup r'$
 si attache$(f) =$ numero(g)
 alors $r \leftarrow r \cup \{g\}$
 sinon attache$(g) \leftarrow \min($attache$(g),$ attache$(f))$
 renvoyer (V, n, r)
fonction INDISPENSABLES(R)
 $g \leftarrow$ un des sommets de R
 numero$(g) \leftarrow 1$
 $f \leftarrow$ un des voisins de g
 $(V, n, r) \leftarrow$ DFS$(\{g\}, 2, f, g)$
 pour chaque h voisin de g
 faire { **si** $h \notin V$
 faire { $(V, n, r') \leftarrow$ DFS(V, n, h, g)
 $r \leftarrow r \cup r' \cup \{g\}$
 renvoyer r

Cet algorithme peut être utilisé directement pour savoir si on peut retirer une voie. Mais il se pose aussi la question de savoir si au départ le réseau était assez redondant, et au cas où il

contenait une station indispensable, comment rajouter des voies pour y remédier. Pour borner le nombre de voies à rajouter, on compte le nombre de sous-ensembles du réseau qui soient sûrs. Formellement, on appelle sous-réseau sûr un sous-ensemble des stations du réseau tel que si on considère le sous-graphe formé par ces stations, aucune station n'est indispensable, et si on rajoute une station du réseau global, alors apparaît forcément une station indispensable.

Question 15. Modifiez votre algorithme pour qu'il retourne un ensemble de voies au plus aussi grand que le nombre de sous-réseaux du réseau sûrs, et tel que si on ajoute ces voies au graphe, il ne contienne plus de station indispensable. Vous justifierez la correction de votre algorithme.

Corrigé. Une station indispensable est un passage obligé entre deux sous-ensembles du réseau sûrs maximaux. Le père de la station indispensable appartient forcément à un sous-ensemble distinct de chacun des fils de cette station dont le point d'attache est cette station. De plus, si on rajoute une voie entre le père et chacun de ces fils, alors cette station ne peut plus être indispensable.

fonction DFS$(V, n, g, pere)$
 si $g \in V$ **renvoyer** $(V, n, \emptyset)$
 numero$(g) \leftarrow n$
 $n \leftarrow n + 1$
 $V \leftarrow V \cup \{g\}$
 attache$(g) \leftarrow$ numero$(pere)$
 $r \leftarrow \emptyset$ — Les voies à rajouter
 pour chaque f voisin de g
  faire { **si** $f \in V$ **alors** attache$(g) \leftarrow \min($attache$(g),$ numero$(f))$
  sinon { $(V, n, r') \leftarrow$ DFS(V, n, f, g); $r \leftarrow r \cup r'$; **si** attache$(f) =$ numero(g) **alors** $r \leftarrow r \cup \{pere, f\}$ **sinon** attache$(g) \leftarrow \min($attache$(g),$ attache$(f))$
 renvoyer (V, n, r)
fonction CompleteReseau(R)
 $g \leftarrow$ un des sommets de R
 numero$(g) \leftarrow 1$
 $f \leftarrow$ un des voisins de g
 $(V, n, r) \leftarrow$ DFS$(\{g\}, 2, f, g)$
 pour chaque h voisin de g
  faire { **si** $h \notin V$ **faire** { $(V, n, r') \leftarrow$ DFS(V, n, h, g); $r \leftarrow r \cup r' \cup \{(f, h)\}$
 renvoyer r