Devoir surveillé n° 06 — corrigé

1 Logique

Q 1) Je propose

- b : « je marque un but »
- c : « je suis content »
- f : « je fais la fête »
- g : « mon équipe gagne »

Q 2) On trouve les formules suivantes que l'on met sous forme normale conjonctive.

i) $b \to c \wedge f \equiv (\neg b \vee c) \wedge (\neg b \vee f)$

ii) $g \to c \vee f \equiv \neg g \vee c \vee f$

iii) $\neg g \to \neg c \vee b \equiv g \vee \neg c \vee b$

iv) $\neg b \wedge f \to c \equiv b \vee \neg f \vee c$

v) $\neg c$

Q 3) Cf. question précédente.

Q 4) On considère la formule résultant de la conjonction des 5 formules ci-dessus

$$F = (\neg b \vee c) \wedge (\neg b \vee f) \wedge (\neg g \vee c \vee f) \wedge (g \vee \neg c \vee b) \wedge (b \vee \neg f \vee c) \wedge \neg c$$

On obtient une formule en forme normale conjonctive dont on cherche un modèle. Le problème est donc celui de la satisfiabilité d'une formule en forme normale conjonctive et d'en trouver un modèle s'il existe.

Q 5) L'algorithme de Quine ne précise pas l'ordre dans lequel on choisit les variables, on va prendre un ordre qui nous arrange.

- On choisit c. On a $C_1 = C[c \leftarrow \bot] = \{\neg b, \neg b \vee f, \neg g \vee f, b \vee \neg f\}$ et on cherche $Quine(C_1)$.
- On choisit b, on a $C_2 = C_1[b \leftarrow \bot] = \{\neg g \vee f\ \neg f\}$ et on cherche $Quine(C_2)$.
- On choisit f. On a $C_3 = C_2[f \leftarrow \top] = \{\neg g\}$ et on cherche $Quine(C_3)$.
- On choisit g. On a $C_4 = C_3[g \leftarrow \bot] = \varnothing$ donc $Quine(C_4) = Vrai$.

Remarquons que l'on a exploré uniquement la branche de gauche, en suivant scrupuleusement l'algorithme de Quine. L'algorithme renvoie donc *Vrai* pour la formule F et l'on peut en déduire que la valuation constante égale à faux est un modèle de F. Une possibilité pour le joueur est donc de ne pas avoir marqué, d'avoir perdu, d'être triste et de ne pas faire la fête. On peut d'ailleurs vérifier que c'est la seule.

2 Bases de données

Q 6) Je propose le modèle relationnel suivant. Les clés primaires sont en gras souligné, les clés étrangères en italique et précèdent la flèche qui pointe vers la table référencée.

- **Ville** : (**Code_Postal**, **Nom_Ville**)
- **Cinéma** : (**Code_Cine**, Nom, Adresse, *Code_Postal*, *Nom_Ville* → *Ville*)
- **Salle** : (**Code_Salle**, Capacité, 3D, *Code_Cine* → *Cinéma*)
- **Projette** : (***Code_Film*** → *Film*, ***Code_Salle*** → *Salle*, nb_Entrées)
- **Film** : (**Code_Film**, Titre, Durée, *Code_Réal* → *Réalisateur*)

- **Réalisateur** : (**Code_Réal**, Nom_Réal, Prénom_Réal, Âge)

☞ **Remarque 1**

- *L'énoncé précise bien que la clé primaire pour la relation* ***Ville*** *doit être constituée des deux attributs* Code_Postal *et* Nom*, ce qui n'incite pas à ajouter un identifiant unique. On utilise alors cette clé formée des deux attributs comme clé étrangère dans la table* ***Cinéma****.*
- *Les cardinalités* $(1, n)$ *sur les deux pattes de l'association* ***Projette*** *imposent d'en faire une relation et pas simplement un lien via les clés étrangères.*
- *Je ne suppose pas qu'un film peut être projeté plusieurs fois dans une même salle avec un nombre d'entrées différentes, mais plutôt que l'on comptabilise une seule fois toutes les entrées d'un film dans une salle. Il suffirait sinon d'ajouter un identifiant pour chaque projection et de l'utiliser comme clé primaire.*
- *On peut imaginer plusieurs réalisateurs de même nom, prénom et âge, aussi j'introduis un identifiant unique* Code_Réal *qui permet de plus de simplifier la clé étrangère dans* ***Film****. Les notations de l'énoncé semblent cependant suggérer que* Nom_Réal *et* Prénom_Réal *identifient un réalisateur.*
- *On pourrait imaginer que le code des salles n'est pas unique pour l'ensemble des cinémas mais seulement au sein de chaque cinéma. Il suffierait alors simplement d'ajouter* Code_Cinéma *à la clé primaire.*

Q 1) On suppose que l'inégalité est large.

SQL
```sql
SELECT Titre FROM Film WHERE Durée <= 2
```

Q 2) On fait une simple jointure.

SQL
```sql
SELECT R.Nom_Réal, R.Prénom_Réal
FROM Réalisateur R JOIN Film F ON R.Code_Réal = F.Code_Real
WHERE F.Titre = 'Matrix'
```

Q 3) Pas besoin de jointure puisque l'on a toute l'information dans la table.

SQL
```sql
SELECT COUNT(*) FROM Cinéma WHERE Nom_Ville = 'Nantes'
```

Q 4) L'adresse d'un cinéma avec plusieurs salles en 3D va ressortir plusieurs fois, aussi on supprime les doublons, mais ceci ne me semble pas impératif.

SQL
```sql
SELECT DISTINCT C.Adresse
FROM Cinéma C JOIN Salle S ON C.Code_Cinema = S.Code_Cinema
WHERE S.3D = 1
```

Q 5) Il s'agit d'une division cartésienne. Il y a plusieurs manières de faire. Celle qui me semble la plus simple à retenir de manière générale est de reformuler sous la forme d'une double négation et d'utiliser `NOT EXISTS`. On peut aussi utiliser `COUNT` si on veut vraiment rester dans le cadre du programme. On cherche les films pour lesquels il n'existe pas de salle pour laquelle il n'existe pas de projection de ce film dans cette salle.

SQL

```sql
SELECT Code_Film FROM Film F
WHERE NOT EXISTS (
  SELECT Code_Salle FROM Salle S
  WHERE NOT EXISTS (
    SELECT * FROM Projette P
    WHERE P.Code_Film = F.Code_Film AND P.Code_Salle = S.Code_Salle
  )
)
```

Q 6) Il faut faire toutes les jointures pour obtenir l'information. On aura des doublons donc j'utilise `DISTINCT`, mais cela ne me semble pas explicitement demandé.

SQL

```sql
SELECT DISTINCT F.Titre
FROM Film F
  JOIN Projette P ON F.Code_Film = P.Code_Film
  JOIN Salle S ON P.Code_Salle = S.Code_Sall
  JOIN Cinéma C ON S.Code_Cine = C.Code_Cine
WHERE C.Nom = 'Le Rio'
```

3 Algorithmique — Problèmes de dominos

3.1 Structures de données

Q 7) Pas de difficulté.

C

```c
struct element {
  Domino domino;
  struct element *suivant;
};
typedef struct element element;
```

Q 8) Une liste chaînée est un pointeur vers son première élément et le pointeur `NULL` pour la liste vide.

C

```c
typedef element* chaine;
```

Q 9) Il est beaucoup plus simple de proposer une version récursive. Comme je vais proposer plusieurs versions, je commence par une fonction qui permet de créer un maillon.

C

```c
element *creerElement(Domino d) {
  element *elt = malloc(sizeof(element));
  elt->domino = d;
  elt->suivant = NULL;
  return elt;
}
```

On procède ensuite récursivement : si la liste est vide on renvoie ce nouveau maillon, sinon, on ajoute dans la queue et on renvoie la tête.

C

```c
element *ajoutElement(element *l, Domino d) {
  if (l == NULL) {
    return creerElement(d);
  } else {
    l->suivant = ajoutElement(l->suivant, d);
    return l;
  }
}
```

On peut également procéder de manière itérative, mais c'est un peu plus délicat.

C

```c
element *ajoutElement(element *l, Domino d) {
  // Si la liste est vide elle devient réduite à un maillon
  if (l == NULL) {return creerElement(d);}
  // Sinon on va chercher le dernier maillon pour accrocher le maillon
  element *dernier = l;
  while (dernier->suivant != NULL) {
    dernier = dernier->suivant;
  }
  dernier->suivant = creerElement(d);
  return l;
}
```

Q 10) Il y a une première subtilité. On ne peut pas utiliser l'opérateur == pour comparer deux structures pour de sombres histoires d'alignement hors programme, il nous faut donc implémenter une fonction pour cela. Il n'est pas clair dans le sujet si un domino est défini à rotation près ou non à ce stade. Comme nous aurons besoin plus tard d'une égalité à rotation près, on la propose dès maintenant.

C

```c
bool eqDomino(Domino d1, Domino d2) {
  return (d1.x == d2.x && d1.y == d2.y) ||
         (d1.x == d2.y && d1.y == d2.x);
}
```

Il n'y a pas d'hypothèses sur la présence ou non du domino dans la liste. On ne va pas supposer que celui-ci est présent, mais on ne supprime que sa première occurrence si celui-ci est présent plusieurs fois (ce qui semble possible pour une chaîne mais pas pour un sac). On procède récursivement : si la liste est vide il n'y a rien à faire ; si le domino est en première position on renvoie la queue de la liste, sans oublier de libérer la mémoire ; sinon on supprime récursivement l'élément dans la queue.

C

```c
element *retireElement(element *l, Domino d) {
  if (l == NULL) {
    return NULL;
  } else if (eqDomino(l->domino, d)) {
    element* suivant = l->suivant;
    free(l);
    return suivant;
  } else {
    l->suivant = retireElement(l->suivant, d);
    return l;
  }
}
```

On peut donner une version itérative bien plus délicate. On cherche le maillon précédent celui qui contient le domino pour pouvoir supprimer le maillon suivant. Attention à traiter correctement le cas où le domino était en tête de la liste.

C
```c
element *retireElement(element *l, Domino d) {
  element *precedent = NULL;
  element *curseur = l;
  while (curseur != NULL) {
    if (eqDomino(curseur->domino, d)) {
      if (precedent != NULL) {
        precedent->suivant = curseur->suivant;
      } else {
        l = curseur->suivant;
      }
      free(curseur);
      return l;
    }
    precedent = curseur;
    curseur = curseur->suivant;
  }
  return l;
}
```

Q 11) La version récursive s'écrit en une ligne.

C
```c
bool rechercheElement(element *l, Domino d) {
  return l != NULL && (eqDomino(l->domino, d) ||
                       rechercheElement(l->suivant, d));
}
```

La version itérative n'est pas beaucoup plus compliquée.

C
```c
bool rechercheElement(element *l, Domino d) {
  while (l != NULL) {
    if (eqDomino(l->domino, d)) {return true;}
    l = l->suivant;
  }
  return false;
}
```

3.2 Existence d'une chaîne de taille n

Q 12) Cette question est incompréhensible. On croit comprendre qu'il faut aller chercher les dominos i et j dans un sac qui n'est pas passé comme argument, ce qui serait de toute façon d'une réelle inefficacité. On va plutôt supposer qu'il s'agit d'écrire une fonction qui prend en paramètres deux dominos D_i et D_j et qui vérifie s'il est possible de placer le domino D_j à *droite* (et non à *gauche*) du domino D_i.

C
```c
bool possibleSansRotation(Domino Di, Domino Dj) {
  return Di.y == Dj.x;
}
```

Q 13) Je suppose que l'idée est d'effectuer la rotation sur le domino D_j si celle-ci est nécessaire pour pouvoir le placer à droite de D_i. Pour cela il faut effectivement passer le domino par adresse pour pouvoir agir sur celui-ci par effet de bord.

Q 14) La spécification de cette fonction n'est pas donnée. On va supposer qu'elle renvoie `true` tout en effectuant la rotation si et seulement si on peut placer D_j à droite de D_i après rotation.

C
```c
bool possibleAvecRotation(Domino Di, Domino *Dj) {
  if (Di.y == Dj->y) {
    int tmp = Dj->x;
    Dj->x = Dj->y;
    Dj->y = tmp;
    return true;
  } else {
    return false;
  }
}
```

Q 15) Pour passer d'une k-chaîne à une $k+1$-chaîne (sans doublons), on rajoute un domino parmi ceux non utilisés dans la k-chaine.

Q 16) On adopte une démarche de retour sur trace qui, à partir d'une k-chaîne, indique s'il existe une chaîne qui complète cette chaîne en utilisant tous les dominos (une seule fois chacun). À partir d'une k-chaîne, on essaie successivement un par un tous les dominos non utilisés dans la k-chaîne pour constuire une $k+1$-chaîne et essayer, par un appel récursif, de la compléter en une n-chaîne. Si on y parvient pour un des dominos, on renvoie VRAI et sinon on renvoie FAUX. Il suffit ensuite de lancer l'exploration à partir de la 0-chaîne.

Q 17) Cette question me semble excessivement difficile sans indications. Il suffit de suivre l'idée de la question précédente mais il y a de nombreux points très délicats dans l'implémentation, surtout avec les structures de données suggérées. Voici une proposition, certainement pas optimale, en cherchant à utiliser les fonctions précédentes. On propose tout d'abord deux fonctions auxiliaires.

C
```c
element *copie(element *src) {
  element *dst = NULL;
  while (src != NULL) {
    dst = ajoutElement(dst, src->domino);
    src = src->suivant;
  }
  return dst;
}
```

C
```c
void free_list(element *lst) {
  if (lst != NULL) {
    free_list(lst->suivant);
    free(lst);
  }
}
```

Écrivons maintenant une fonction qui considère une k-chaîne terminée par un certain domino et qui cherche à la prolonger en utilisant les dominos restants.

C

```c
/* Essaie de compléter la chaîne de dominos `*chaine`, qui se termine
par le domino `dernier`, avec tous les dominos restant dans le sac de
dominos `*sac`. À tout instant la chaîne et le sac partitionnent
l'ensemble de tous les dominos disponibles. */
element *complete_chaine(element **chaine, Domino dernier, element **sac) {
  // S'il n'y a plus de dominos dans le sac on a une n-chaine
  // et c'est gagné
  if (*sac == NULL) {
    return *chaine;
  }
  // Remplacé par la chaîne complète si on en trouve une
  element *res = NULL;
  // Sinon, on essaie chaque domino restant dans le sac. On créé une
  // copie des dominos du sac (dont l'ordre va être modifié) pour ne
  // pas avoir de problème
  element *copie_sac = copie(*sac);
  // On considère ces éléments un par un
  element *elt = copie_sac;
  while (elt != NULL) {
    if (possibleAvecRotation(dernier, &elt->domino)) {
      // On peut bien placer ce domino en fin de chaîne,
      // éventuellement inversé
      *chaine = ajoutElement(*chaine, elt->domino);
      *sac = retireElement(*sac, elt->domino);
      res = complete_chaine(chaine, elt->domino, sac);
      if (res != NULL) {
        break;
      }
      // Il ne faut pas oublier de remettre le domino dans le sac, il
      // sera placé à la fin, mais l'ordre n'a pas d'importance
      *chaine = retireElement(*chaine, elt->domino);
      *sac = ajoutElement(*sac, elt->domino);
    }
    elt = elt->suivant;
  }
  free_list(copie_sac);
  return res;
}
```

Écrivons une fonction qui permet d'effectuer la rotation.

C

```c
Domino rotation(Domino d) {
  Domino r = {.x = d.y, .y = d.x};
  return r;
}
```

Et enfin la fonction principale.

C

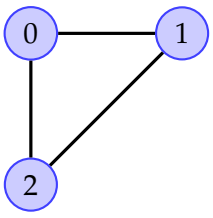
```c
element *trouve_chaine(element *sac) {
  element *chaine = NULL;
  element *copie_sac = copie(sac);
  element *res = NULL;
  // On essaie avec chaque début possible dans un sens ou l'autre
  element *elt = sac;
  while (elt != NULL) {
    chaine = ajoutElement(chaine, elt->domino);
    copie_sac = retireElement(copie_sac, elt->domino);
    res = complete_chaine(&chaine, elt->domino, &copie_sac);
    if (res != NULL) {break;}
    chaine->domino = rotation(elt->domino);
    res = complete_chaine(&chaine, rotation(elt->domino), &copie_sac);
    if (res != NULL) {break;}
    chaine = retireElement(chaine, elt->domino);
    copie_sac = ajoutElement(copie_sac, elt->domino);
    elt = elt->suivant;
  }
  free_list(copie_sac); // S'il reste des choses
  return res;
}
```

Q 18) Dans le pire cas on explore entièrement l'arborescence de recherche qui comporte $2^n n!$ feuilles correspondant aux $n!$ permutations possibles où chaque domino peut être inversé où non. Pour chacun des $O(2^n n!)$ nœuds de l'arbre on effectue $O(n)$ opérations (insertion en fin d'une liste de taille au plus n, etc.). La complexité est donc en $O(n2^n n!)$.

3.3 Nombre de chaînes de taille n

Q 19) Il y a $N+1$ dominos doubles et on choisit deux éléments parmi $N+1$ sans répétitions pour former les dominos avec deux faces différentes. On a donc $n = |S_N| = \binom{N+1}{1} + \binom{N+1}{2} = \frac{(N+1)(N+2)}{2}$

Q 20) Il s'agit du graphe complet à trois sommets :



Q 21) Chaque sommet de K_{N+1} est relié à tous les N autres sommets et est donc de degré N.

Q 22) Chaque arête possède deux extrémités et chaque sommet est extrémité de $d(s)$ arête, où l'on note $d(s)$ le degré d'un sommet $s \in S$. En dénombrant de ces deux manières le nombre d'extrémités, on a la relation suivante :

$$\sum_{s \in S} d(s) = 2|A|$$

Une somme paire de nombre entiers contient un nombre pair d'entiers impairs. Le nombre de sommets de degré impair est donc pair.

Q 23) Soit $G = (S, A)$ un graphe possédant un chemin eulérien et soit $c = s_0 s_1 \dots s_p$ un tel chemin. Pour tout $i \in [\![1, p-1]\!]$, le sommet s_i est incident aux deux arêtes $\{s_{i-1}, s_i\}$ et $\{s_i, s_{i+1}\}$. Chaque sommet $s \in S \setminus \{s_0, s_p\}$ est donc incident à exactement deux arêtes pour chaque occurrence de ce sommet dans le chemin et comme toutes les arêtes apparaissent exactement une fois, toutes ces arêtes sont deux à deux distinctes et constituent exactement les arêtes incidentes à ce sommet. Tous les sommets,

sauf éventuellement deux, ont donc un degré pair. Remarquons que s'il y a des sommets de degré impair, il y en a nécessairement exactement deux et ce sont les deux extrémités du chemin eulérien, et que sinon le chemin est un cycle eulérien.

Q 24) Supposons tout d'abord que s_1 et s_2 ne sont pas reliés dans G. Considérons le graphe $G' = (S, A')$ avec $A' = A \cup \{s_1, s_2\}$. Le degré de s_1 et s_2 dans G' est pair et celui de tous les autres sommets est inchangé donc reste pair. Par hypothèse, G' comporte donc un chemin eulérien qui, comme on l'a vu dans la question précédente, est un cycle eulérien. En supprimant l'arête $\{s_1, s_2\}$ de ce cycle on obtient un chemin eulérien dans G. Supposons maintenant que s_1 et s_2 sont reliés dans G et considérons $G' = (S, A')$ avec $A' = A \setminus \{s_1, s_2\}$ dont tous les sommets sont de degré pair. Si G' est connexe il admet par hypothèse un cycle eulérien $c = s_1, \ldots, s_1$ que l'on peut compléter avec l'arête pour obtenir un chemin eulérien $c' = s_1, \ldots, s_1, s_2$ de G. Sinon, c'est que $\{s_1, s_2\}$ est un isthme de G et G' comporte alors deux composantes connexes G_1 et G_2 auxquelles on peut appliquer l'hypothèse pour obtenir deux cycles $c_1 = s_1 \ldots s_1$ de G_1 et $c_2 = s_2 \ldots s_2$ de G_2. On peut réunir ces deux cycles par l'arête $\{s_1, s_2\}$ pour obtenir un chemin eulérien $c = s_1, \ldots, s_1, s_2, \ldots, s_2$ de G.

Q 25) Si $C = s_1, \ldots, s_2$ avec $s_1 \neq s_2$ est ouvert, alors s_1 est incident à un nombre impaire d'arêtes de C. Comme s_1 est de degré pair dans G, il existe une arête de G qui n'est pas dans C qui permettrait de prolonger C ce qui est absurde.

Q 26) Parmi toutes les arêtes qui ne sont pas dans C, il en existe au moins une incidente à un sommet de C, puisque le graphe est connexe. Notons $\{s, s'\}$ une telle arête avec $s \in C$. C est fermé et peut s'écrire $s, \ldots, s$. Le chemin $s, \ldots, s, s'$ est alors strictement plus long que C ce qui est encore absurde.

Q 27) Dans K_7 tous les sommets sont de degré 6 et donc pairs. K_7 possède donc des cycles eulériens (et donc des chemins eulériens).

Q 28) La formule proposée n'est pas valable pour $N \in \{0, 1\}$. On suppose que $N \geq 2$ et on admet le résultat implicite dans l'énoncé : les chaines de $n - N - 1$ dominos distincts non doubles sont en bijection avec les chemins eulériens de K_{N+1}. Si N est impair, comme $N + 1 \geq 3$ et que tous les sommets sont de degré N impair, il n'existe pas de chemin eulérien dans K_{N+1} et donc pas de chaîne de tous les dominos sans doubles, et donc pas de chaîne de n dominos avec les doubles non plus. La formule proposée vaut bien 0 puisque $E_{N+1} = 0$. Si N est pair, alors K_{N+1} comporte $E_{N+1} > 0$ circuits eulériens et pas de chemin eulérien qui ne soit pas un circuit. Chacun de ces circuits eulériens correspond à un circuit de dominos sans doubles. Reste à placer les $N + 1$ dominos doubles dans ce circuit de dominos. Puisque tous les sommets de K_{N+1} sont de degré N, chaque sommet a exactement $N/2$ occurrences dans le circuit, qui correspondent précisément aux positions où il est possible de placer le domino double correspondant. Il y a donc $\left(\frac{N}{2}\right)^{N+1}$ manières de placer les $N + 1$ dominos doubles. Chaque circuit de longueur n correspond à n chaînes correspondant chacune au choix d'un domino de départ. Il y a donc $n\left(\frac{N}{2}\right)^{N+1} E_{N+1}$ chaînes des n dominos, ce qui est la formule proposée.

3.4 Recherche de la plus longue sous-séquence

Q 29) On note $D_{i-1} \sim D_i$ si on peut placer le domino D_i à droite du domino D_{i-1}. On a alors :

$$l(i) = \begin{cases} 1 + l(i-1) & \text{si } i > 1 \text{ et } D_{i-1} \sim D_i \\ 1 & \text{sinon} \end{cases}$$

Q 30) Il suffit d'initialiser un tableau pour l que l'on rempli de gauche à droite.

Q 31) On a $l_{1,0} = l_{1,1} = 1$.

Q 32) On peut ajouter un champ formé d'un booléen indiquant le sens du domino.

Entrée : $D_1, D_2, \ldots, D_n$
$l \leftarrow \{1, 1, \ldots, 1\}$;
pour i *de* 2 *à* n **faire**
  si $D_{i-1} \sim D_i$ **alors**
    $l[i] \leftarrow l[i] + l[i-1]$;
  fin
fin
Renvoyer : $\max(l)$

C

```c
struct element {
  Domino domino;
  bool direct;
  struct element *suivant;
};
```

Q 33) On note $\widetilde{D}_i$ le domino D_i après rotation. On a, pour $i \in [\![2, n]\!]$:

$$l_{i,0} = \max\left\{\begin{cases} 1 + l_{i-1,0} & \text{si } D_{i-1} \sim D_i \\ 1 \text{ sinon} \end{cases}, \begin{cases} 1 + l_{i-1,1} & \text{si } \widetilde{D}_{i-1} \sim D_i \\ 1 \text{ sinon} \end{cases}\right\}$$

$$l_{i,1} = \max\left\{\begin{cases} 1 + l_{i-1,0} & \text{si } D_{i-1} \sim \widetilde{D}_i \\ 1 \text{ sinon} \end{cases}, \begin{cases} 1 + l_{i-1,1} & \text{si } \widetilde{D}_{i-1} \sim \widetilde{D}_i \\ 1 \text{ sinon} \end{cases}\right\}$$

Il reste à calculer les deux suites $l_i, 0$ et $l_i, 1$ et prendre le maximum de toutes les valeurs trouvées pour tous les $i \in [\![1, n]\!]$ pour trouver la plus longue sous-séquence.