Master DAPM - Session 1 - Complexité Algorithmique

Lundi 02/03/2015. 14h00-17h00. Salle T' 302

La précision et la clarté de votre rédaction sont *fondamentales*. Chaque réponse doit être accompagnée d'une justification. Lisez attentivement l'énoncé avant de commencer. Le barème est indicatif. Documents interdits. Durée 3 heures.

L'examen a finalement été noté sur 25 points.

Exercice 1. [3 pts] Rappelez la définition d'une suite. Démontrez que l'ensemble des suites binaires n'est pas dénombrable. Indication : par l'absurde, rangez les termes de ces suites dans une table et construisez une suite (u_n) différente de toutes ces suites.

Solution de l'exercice 1. Une suite est une application de l'ensemble des entiers naturels $\mathbf{N}$ dans un ensemble quelconque. Une suite binaire est tout simplement une application de $\mathbf{N}$ dans l'ensemble $\{0,1\}$. Par l'absurde, supposons que l'ensemble des suites binaires, que l'on notera B, soit dénombrable. On dispose alors (par définition d'un ensemble dénombrable) d'une bijection $b : \mathbf{N} \to B$ et par conséquent d'une suite $(b_n)_{n\in\mathbf{N}}$ d'éléments deux-à-deux distincts de B telle que $B = \{b_n \mid n \in \mathbf{N}\}$. Soient i et j deux entiers naturels, on note $b_{i,j}$ le j-ème bit de la i-ème suite b_i de B :

$$\begin{array}{lccccccc}
\boldsymbol{b}_0 : & \boxed{b_{0,0}} & b_{0,1} & b_{0,2} & b_{0,3} & \cdots & \cdots \\
\boldsymbol{b}_1 : & b_{1,0} & \boxed{b_{1,1}} & b_{1,2} & b_{1,3} & \cdots & \cdots \\
\boldsymbol{b}_2 : & b_{2,0} & b_{2,1} & \boxed{b_{2,2}} & b_{2,3} & \cdots & \cdots \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
\boldsymbol{b}_n : & b_{n,0} & b_{n,1} & b_{n,2} & \cdots & \boxed{b_{n,n}} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

Il suffit à présent de construire une suite binaire (u_n) qui ne soit pas dans cette liste ce qui est bien sûr contradictoire puisque cette liste est sensée les contenir toutes. La suite binaire (u_n) de terme général $u_n := 1 - b_{n,n}$ est différente de n'importe quelle suite (b_k) puisque deux suites (u_n) et (v_n) ne peuvent être égales que si $\forall n \in \mathbf{N},\ u_n = v_n$.

Exercice 2. [5 pts] Soit $f : \mathbf{N} \to \mathbf{N}$ une fonction Turing-calculable avec l'alphabet unaire $\Sigma := \{!\}$. On suppose que l'algorithme qui la calcule n'utilise aucune des cellules à gauche des $n+1$ bâtons qui représentent l'entier n. On note k le nombre de ses états et on suppose que l'algorithme s'arrête dans l'état q_{k-1}. Par convention, la tête de lecture-écriture est placée sur le bâton le plus à gauche au début et la fin de l'exécution.

1. [1 pt] Démontrez que la fonction de transition δ n'est pas définie en $(q_{k-1}, !)$.

2. [4 pts] Démontrez que la fonction $g : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ définie par $g(r,n) := f^r(n)$ est Turing-calculable où f^r est la composée de r fois f : $f \circ f \circ \cdots \circ f$. Expliquez votre algorithme en détail et explicitez les blocs d'instructions de votre algorithme.

Solution de l'exercice 2. Par définition la machine de Turing s'arrête quand sa fonction de transition n'est pas définie en le couple (q, α) constitué par l'état et le symbole courants respectivement. Comme par hypothèse la machine s'arrête sur le bâton PAG [1] dans l'état q_{k-1}, elle n'est pas définie en $(q_k, !)$ (oui la question était triviale).

Nous allons montrer que la fonction g est Turing-calculable avec l'algorithme suivant :

(1) On efface le premier des $r+1$ bâtons représentant r ;

(2) On se déplace d'une case à droite sur le premier des r bâtons restants ;

(3) TANT QU'il reste un bâton :
- On l'efface ;
- On déplace la tête de lecture-écriture sur le bâton PAG de la séquence qui représente $f^{k-1}(n)$ à la k-ème itération de la boucle, i.e. n quand $k = 1$;
- On calcule f en la valeur $f^{k-1}(n)$ à la k-ème itération de la boucle [2] ;
- On ramène la tête de lecture-écriture sur le bâton PAG de la séquence de gauche qui représente $r - k$ à la k-ème itération de cette boucle ;

(4) On finit en déplaçant la tête de lecture-écriture sur le bâton PAG de $f^r(n)$.

Plutôt que de renuméroter tous les états de la machine qui calcule f et pour plus de lisibilité, on préfixera les états à rajouter par la lettre r, le nouvel état initial sera par convention r_0. Les instructions à rajouter au programme qui calcule f pour calculer g sont les suivantes :

$$\begin{array}{rcll}
(r_0, !) & \Rightarrow & (\square, r_1) & \text{DEBUT : on efface le premier des } r+1 \text{ bâtons puis} \\
(r_1, \square) & \Rightarrow & (\rightarrow, r_2) & \text{on va à droite et on teste s'il reste un bâton :} \\
(r_2, \square) & \Rightarrow & (\rightarrow, r_7) & \textit{Il ne reste aucun bâton} \text{ : on avance sur le blanc séparateur,} \\
(r_7, \square) & \Rightarrow & (\rightarrow, r_8) & \text{puis à droite sur le bâton PAG de } f^r(n), \text{ on s'arrête FIN.} \\
(r_2, !) & \Rightarrow & (\square, r_3) & \textit{Il reste un bâton} \text{ : on l'efface puis} \\
(r_3, \square) & \Rightarrow & (\rightarrow, r_4) & \text{on va à droite et} \\
(r_4, !) & \Rightarrow & (\rightarrow, r_4) & \text{on saute tous les bâtons vers la droite puis} \\
(r_4, \square) & \overset{(A)}{\Rightarrow} & (\rightarrow, \boldsymbol{q}_0) & \text{le blanc séparateur. On calcule } f \text{ en } f^{k-1}(n). \\
(\boldsymbol{q}_{k-1}, !) & \overset{(B)}{\Rightarrow} & (<, r_5) & \text{Fin du calcul de } f, \text{ on se déplace à gauche,} \\
(r_5, \square) & \Rightarrow & (<, r_6) & \text{on recule sur le blanc séparateur et} \\
(r_6, !) & \Rightarrow & (<, r_6) & \text{on saute tous les bâtons vers la gauche pour se placer} \\
(r_6, \square) & \Rightarrow & (>, r_2) & \text{sur le bâton PAG qui reste de } r. \text{ On boucle sur le test } r_2.
\end{array}$$

1. le plus à gauche
2. à la première itération $k = 1$ et f^0 est la fonction identité

Le programme calculant f prend le relais après l'instruction (A). L'instruction (B) est valide d'après la question 1. Nous pouvons utiliser la partie gauche de la bande pour ranger les bâtons représentant l'entier r puisque cette partie est inutilisée par hypothèse.

Exercice 3. [9 pts] On se propose de démontrer que l'*addition* $f : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ définie par $f(m, m) := m + n$ est Turing-calculable avec l'alphabet binaire $\Sigma := \{0, 1\}$. Les entiers m et n sont écrits en binaire, bit de poids fort à gauche et sont séparés par une case vide. La tête de lecture-écriture est placée sur le bit de poids fort de m au départ.

1. [2 pts] Quelles sont les instructions qui réalisent l'addition de deux bits sans retenue ? Les 4 entrées possibles sur la bande et le résultat de l'exécution sont résumés ci-dessous (la case soulignée indique la position de la tête de lecture-écriture) :

$$\underline{0}\square 0 \;\Rightarrow\; \underline{0} \qquad \underline{0}\square 1 \;\Rightarrow\; \underline{1} \qquad \underline{1}\square 0 \;\Rightarrow\; \underline{1} \qquad \underline{1}\square 1 \;\Rightarrow\; \underline{0}$$

2. [5 pts] Généralisez le calcul de l'addition sur des entiers m et n arbitrairement longs. Indication : s'il y a retenue, il suffit d'adapter le bloc de la question 1 en écrivant 1 à la place de 0 et réciproquement. Décrivez *clairement* votre algorithme et précisez le rôle des blocs d'instructions qui constituent votre programme.

3. [2 pts] Calculez la fonction de complexité $t(m, n)$ de votre machine, i.e. le nombre de transitions appliquées pour l'entrée (m, n).

Solution de l'exercice 3. Pour alléger les programmes dans cet exercice, l'écriture $(q, a|b) \Rightarrow (q', c)$ résumera les deux instructions $(q, a) \Rightarrow (q', c)$ et $(q, a) \Rightarrow (q', c)$.

1. Pour calculer $m + n$ sur 1 bit sans retenue, on peut se contenter de changer le bit représentant m dans le cas ou le bit de n est égal à 1.

$(q_0, 0\|1)$	$\Rightarrow\ (\rightarrow, q_1)$	DÉBUT : on saute le bit de m à droite puis
$(q_1, \square)$	$\Rightarrow\ (\rightarrow, q_1)$	le blanc séparateur et on teste n :
$(q_1, 0)$	$\Rightarrow\ (\square, q_2)$	$n = 0$: on efface n puis
$(q_2, \square)$	$\Rightarrow\ (\leftarrow, q_3)$	on recule sur le blanc séparateur
$(q_3, \square)$	$\Rightarrow\ (\leftarrow, q_6)$	on recule à nouveau sur m, FIN
$(q_1, 1)$	$\Rightarrow\ (\square, q_4)$	$n = 1$: on efface n puis
$(q_4, \square)$	$\Rightarrow\ (\leftarrow, q_5)$	on recule sur le blanc séparateur
$(q_5, \square)$	$\Rightarrow\ (\leftarrow, q_5)$	on recule à nouveau sur m
$(q_5, 0)$	$\Rightarrow\ (1, q_6)$	$m = 0$: on écrit 1 et on s'arrête, FIN
$(q_5, 1)$	$\Rightarrow\ (0, q_6)$	$m = 1$: on écrit 0 et on s'arrête, FIN

2. Pour le cas général, une première approche possible serait de calquer l'addition comme on la ferait *à la main*. Elle consiste à écrire les bits de $m + n$ à gauche de m à partir des résultats binaires des additions successives bit-à-bit de m et n en partant de leurs bits de poids faible (que l'on efface une fois lus). Cette méthode est fastidieuse car elle implique de caractériser les 8 différentes combinaisons possibles déterminées par le bit de poids faible de m, celui de n et par le bit de retenue. Ceci impose de répliquer de nombreux blocs d'instructions qui réalisent la même chose (en particulier le déplacement de la tête de lecture-écriture du résultat partiel de $n + m$ vers les prochains bits à analyser de m et n et réciproquement) mais qui dépendent de conditions initiales différentes.

Une autre approche consiste à rajouter 1 à m autant de fois que l'on peut retirer 1 à n. Commençons par étudier les algorithmes d'incrémentation et de décrémentation. Dans les deux cas, on suppose que la tête de lecture-écriture est initialement placée sur le *bit de poids faible* de l'entier sur la bande, donc *à droite*. L'algorithme d'incrémentation consiste à remplacer tous les bits à 1 par des 0 jusqu'au premier bit nul rencontré, ou au pire la case vide, qu'il faut alors remplacer par 1 :

(1) TANT QUE la case courante contient 1
- écrire 0 ;
- déplacer la tête de lecture-écriture sur le bit suivant à gauche ;

(2) la case courante est vide ou contient 0, écrire 1.

Les états du sous-programme correspondant sont préfixés par i.

$(i_0, 0\|\square)$	$\Rightarrow\ (1, i_1)$	DEBUT *bit* $= 0$ *ou case vide* $\square$: on remplace par 1,
$(i_1, 1\|0)$	$\Rightarrow\ (\rightarrow, i_1)$	on saute tous les bits vers la droite et
$(i_1, \square)$	$\Rightarrow\ (\square, i_3)$	on s'arrête, FIN.
$(i_0, 1)$	$\Rightarrow\ (0, i_2)$	DEBUT *bit* $= 1$: on remplace par 0,
$(i_2, 0)$	$\Rightarrow\ (\leftarrow, i_0)$	on va à gauche et on recommence le test i_0.

Notons que ce sous-programme ramène la tête de lecture-écriture sur la première case vide à *à droite* de l'entier en binaire une fois l'incrémentation achevée. L'algorithme termine son exécution dans l'état i_3 en lequel la fonction de transition n'est jamais définie.

L'algorithme de décrémentation est à peine plus compliqué. Dans le cas général, il suffit d'intervertir les rôles de 0 et de 1 dans le programme d'incrémentation. Cependant il faut également tenir compte de deux cas particuliers :

(1) si l'entier est égal à 0 alors il faut s'arrêter ;

(2) si l'entier est du type 1.0^k alors il faut le transformer en 1^k ce qui impose d'effacer le bit de poids fort.

Cette fois les états sont préfixés par d. Le premier bloc ci-dessous gère le cas où l'entier est égal à 0, le cas général est traité par l'état d_2 plus loin :

$(d_0, 0)$	$\Rightarrow\ (\leftarrow, d_1)$	*bit* $= 0$: on va à gauche et on teste
$(d_1, \square)$	$\Rightarrow\ (\square, d_7)$	*case vide* : l'entier est nul, on s'arrête, FIN.
$(d_1, 0\|1)$	$\Rightarrow\ (\rightarrow, d_2)$	*case non-vide* : retour au 1er bit pour cas général en d_2.

On est revenu sur le bit de poids faible de l'entier, on traite le cas général :

$(d_2, 0) \Rightarrow (1, d_3)$ $\quad bit = 0$: écrire 1 puis
$(d_3, 1) \Rightarrow (\leftarrow, d_2)$ $\quad$ reculer et recommencer le test d_2 ;
$(d_2, 1) \Rightarrow (0, d_4)$ $\quad bit = 1$: écrire 0 puis
$(d_4, 0) \Rightarrow (\leftarrow, d_5)$ $\quad$ reculer et tester
$(d_5, 0|1) \Rightarrow (\leftarrow, d_8)$ $\quad nombre \neq 1.0^k$: reculer et s'arrêter, FIN ;
$(d_5, \square) \Rightarrow (\rightarrow, d_6)$ $\quad nombre = 1.0^k$: on avance sur le 0 que l'on a écrit ;
$(d_6, 0) \Rightarrow (\square, d_8)$ $\quad$ on l'efface et on s'arrête, FIN.

Notons que cette fois le sous-programme ramène la tête de lecture-écriture sur la première case vide *à gauche* de l'entier en binaire une fois la décrémentation achevée. Cette machine s'arrête dans l'état d_7 si l'entier valait 0 initialement (dans ce cas, on n'a pas décrémenté) et dans l'état d_8 si l'entier était strictement positif (on a décrémenté).

Il ne reste plus qu'à écrire la boucle qui va incrémenter m à chaque fois que l'on aura pu décrémenter n d'une unité :

$(q_0, 0|1) \Rightarrow (\rightarrow, q_0)$ $\quad$ DEBUT : on saute m jusqu'au blanc séparateur à droite ;
$(q_0, \square) \Rightarrow (\rightarrow, q_1)$ $\quad$ on avance pour sauter le blanc séparateur ;
$(q_1, 0|1) \Rightarrow (\rightarrow, q_1)$ $\quad$ on saute tous les bits de n à droite ;
$(q_1, \square) \Rightarrow (\leftarrow, \boldsymbol{d}_0)$ $\quad$ on revient sur le bit de poids faible et on décrémente n ;

Si n a été décrémenté, le sous-programme s'est arrêté sur le blanc séparateur en d_8 :

$(d_8, \square) \Rightarrow (\leftarrow, d_8)$ $\quad$ on saute tous les blancs entre m et n vers la gauche ;
$(d_8, 0|1) \Rightarrow (\rightarrow, \boldsymbol{i}_0)$ $\quad$ on avance sur le bit de poids faible de m et on incrémente ;
$(i_3, \square) \Rightarrow (\rightarrow, q_1)$ $\quad$ on avance sur le bit de poids fort de n et on reprend en q_1 ;

Si la décrémentation n'a pas eu lieu, i.e. la valeur de n a atteint 0, le sous-programme s'est arrêté sur le blanc séparateur en d_7, il faut avancer pour effacer $n = 0$ et revenir sur le bit de poids fort du résultat $n + m$:

$(d_7, \square) \Rightarrow (\rightarrow, q_2)$ $\quad$ on avance sur le 0 ;
$(q_2, 0) \Rightarrow (\square, q_3)$ $\quad$ on l'efface ;
$(q_3, \square) \Rightarrow (\leftarrow, q_3)$ $\quad$ on revient vers m en sautant tous les blancs
$(q_3, 0|1) \Rightarrow (\leftarrow, q_4)$ $\quad$ on recule et
$(q_4, 0|1) \Rightarrow (\leftarrow, q_4)$ $\quad$ on saute tous les bits de $m + n$ vers la gauche ;
$(q_4, \square) \Rightarrow (\rightarrow, q_5)$ $\quad$ on avance sur le bit de poids fort de $m + n$ et on s'arrête, FIN.

3. On note a et b le nombre de chiffres binaires des entiers m et n respectivement. On sait que $a = \lfloor \log_2 m \rfloor + 1$ et $b = \lfloor \log_2 n \rfloor + 1$. On suppose que $m \geq n$, sinon il suffit d'intervertir les rôles de m et n. Chronologiquement :

(1) on déplace la tête vers le bit de poids faible de n, soit $O(a) + O(b)$ transitions ;
(2) à chaque décrémentation de n, on modifie les bits 0 à r où r désigne l'indice du premier bit à 1 en partant du bit de poids faible de n à droite. On pourrait établir un lien entre la valeur de n et cette position r mais on peut éviter ce travail en remarquant que la tête de lecture-écriture doit systématiquement revenir sur le blanc séparateur à gauche du bit de poids fort pour pouvoir ensuite incrémenter m. On a donc $O(b)$ transitions pour une décrémentation.
(3) le nombre de bits modifiés pour une incrémentation de m dépend de la position du premier bit à 0 en partant du bit de poids faible, mais cette fois on ne parcourt pas les bits restants. On peut néanmoins majorer le nombre de bits modifiés par $a + 1$ puisqu'à la fin du traitement l'entier aura pour valeur $m + n$ qui s'écrit au plus sur $a + 1$ bits (on rappelle que $m \geq n$). Comme la tête de lecture-écriture doit revenir sur le blanc séparateur, on a donc une majoration avec $O(2(a + 1)) = O(a)$ transitions.
(4) on déplace la tête du blanc séparateur vers le bit de poids faible de n, soit $O(b)$ transitions ;

Pour une décrémentation on a donc au plus $O(a) + O(b) + O(b) + O(a) + O(b) = O(a) + O(b)$ transitions. Au total il y a m décrémentations, on a finalement au plus $m(O(a) + O(b))$ transitions, soit

$$t(m, n) = O(m \log m \log n).$$

Notons que si l'on avait exprimé la complexité comme il est d'usage en fonction de la taille des données traitées, ici a et b, on aurait eu

$$t(a, b) = O(ab2^b).$$

Exercice 4. [8 pts] On considère les deux problèmes de décision suivants :

Satisfaisabilité à 3 littéraux [3-SAT]
Instance : Un ensemble de n variables booléennes $X = \{x_1, \ldots, x_n\}$ et un ensemble de m clauses $\{C_1, C_2, \ldots, C_m\}$ à 3 littéraux.
Question : Existe-t-il instanciation des n variables de X telle que la conjonction des clauses soit satisfaite ?

Recouvrement des sommets [RS]
Instance : Un graphe non-orienté $G = (X, U)$, une taille $K \in \mathbf{N}$.
Question : Existe-t-il un sous-ensemble $Y \subseteq X$, tel que $\#Y \leq K$ et $\forall (x, y) \in U$, $x \in Y$ ou $y \in Y$?

1. [2pts] Rappelez la définition des classes de langages P, NP et NP-complet et montrez que ces deux problèmes sont dans la classe NP en indiquant la nature de la preuve que l'oracle pourrait fournir et comment la vérifier en temps polynomial.

On transforme une instance de [3-SAT] à n variables et m clauses en instance de [RS] en créant deux types de cliques, celles associées aux variables et celles associées aux clauses :

— pour chacune des n variables x_i, on crée la clique $\{x_i, \overline{x_i}\}$;
— pour chacune des m clauses C_j on crée la clique $\{r_j, s_j, t_j\}$.

Soit

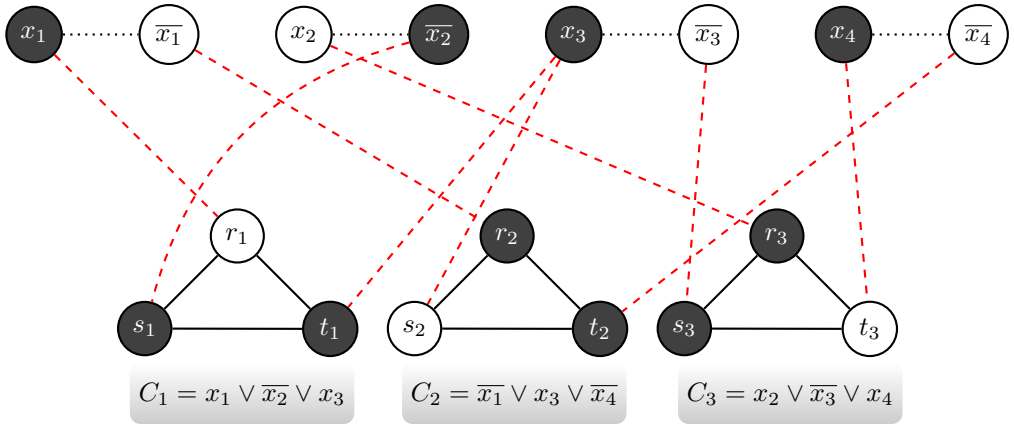
$$X := \bigcup_{i\in[1,n],\ j\in[1,m]} \{x_i, \overline{x_i}, r_j, s_j, t_j\}$$

le graphe contient à ce stade $2n + 3m$ sommets. On achève sa construction en reliant les trois sommets r_j, s_j et t_j de la clique C_j aux littéraux qu'elle contient.

Exemple : le graphe ci-dessous est la transformation de l'instance à 4 variables et 3 clauses suivante :

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee x_4).$$

On fixe la borne K à la valeur $n + 2m$.



2. [2 pts] Montrez que si les clauses sont satisfaisables (par exemple si $x_1, \overline{x_2}, x_3$ et x_4 sont VRAIES), alors il existe un recouvrement Y de taille inférieure ou égale à $K = n + 2m$ sommets. Indication : pour recouvrir les arcs en pointillés, notez que chaque clause contient au moins un littéral VRAI. Compléter le recouvrement Y pour les arcs qui restent (tirets et pleins) avec les sommets de type clause $\{r_j, s_j, t_j\}$ nécessaires.

3. [2 pts] Réciproquement, démontrez que s'il existe un recouvrement Y à $K = n+2m$ sommets alors l'ensemble des clauses C_j peut-être satisfait.

4. [2pts] En admettant que [3-SAT] est NP-complet, montrez que [3-SAT] $\propto$ [RS] et concluez que [RS] est NP-complet.

Solution de l'exercice 4. On rappelle les trois définitions :

1. Un langage est dans la classe P si ses mots sont reconnus par une machine de Turing déterministe polynomiale. Un langage est dans la classe NP si ses mots sont reconnus par une machine de Turing non-déterministe polynomiale. Un langage L est dans la classe NP-complet s'il est dans la classe NP et si tout autre langage L' de la classe NP se transforme polynomialement en L.

Le problème [3-SAT] est dans NP car l'oracle pourrait fournir la table de vérité des variables x_i. L'expression logique contenant exactement $3m$ littéraux, $3m$ recherches dans cette table de vérité ainsi que $3m - 1$ opérations logiques suffiraient à l'évaluer, soit coût linéaire en m si la recherche en table est supposée en temps constant.

Le problème [RS] est dans NP car l'oracle pourrait fournir l'ensemble Y. Il suffirait de vérifier que $\#Y \leq K$ et que chaque arc (x, y) du graphe a l'une de ses extrémités x ou y dans Y, ce qui est linéaire en le nombre d'arcs du graphe.

2. Le graphe contient $2n+3m$ sommets et $n+6m$ arcs par construction. Supposons que l'on dispose d'une instanciation des variables x_i qui satisfasse l'ensemble des clauses C_j. L'ensemble Y de cardinal borné par $K := n+2m$ doit recouvrir trois types d'arcs :

(1) n arcs constituant les n cliques $\{x_i, \overline{x_i}\}$ (pointillés) ;

(2) $3m$ arcs constituant les m cliques $\{r_j, s_j, t_j\}$ (traits pleins) ;

(3) $3m$ arcs reliant les 3 sommets des m cliques $\{r_j, s_j, t_j\}$ aux littéraux (traitillés) ;

On inclut dans Y les n sommets des littéraux VRAIS de chaque clique $\{x_i, \overline{x_i}\}$ qui recouvrent les n arcs $(x_i, \overline{x_i})$ (sommets x_1, $\overline{x_2}$, x_3 et x_4 dans l'exemple).

Deux sommets quelconques d'une clique $\{r_j, s_j, t_j\}$ recouvrent ses 3 arcs mais il faut également recouvrir les 3 arcs qui relient ces 3 sommets aux littéraux associés. Au moins l'un des trois arcs reliant les sommets d'une clique $\{r_j, s_j, t_j\}$ aux littéraux est recouvert par le sommet littéral VRAI associé, sans quoi la clause ne serait pas satisfaite. Il suffit donc de choisir d'intégrer à Y les 2 autres sommets qui ne sont pas connectés à cet arc. Le recouvrement contient exactement $n + 2m$ sommets.

3. Réciproquement, supposons que l'on dispose d'un recouvrement Y de taille $n+2m$. Chaque clique $\{r_j, s_j, t_j\}$ nécessite au moins 2 sommets de Y pour recouvrir ses 3 arcs, il faut donc au moins $2m$ sommets de ces cliques pour toutes les recouvrir. Comme il faut encore au moins n sommets pour recouvrir les n arcs $(x_i, \overline{x_i})$, on n'a pu utiliser plus de 2 sommets de la clique $\{r_j, s_j, t_j\}$ pour la recouvrir. Celui qui n'a pas été utilisé ne peut donc recouvrir l'arc qui l'associe au littéral x_i ou $\overline{x_i}$, c'est donc nécessairement ce littéral qui a été utilisé pour le recouvrir ainsi que l'arc $(x_i, \overline{x_i})$. C'est ce littéral qu'il faut instancier à VRAI pour satisfaire la clause.

4. La transformation d'une instance de [3-SAT] contenant n variables et m clauses crée $n+3m$ sommets et $n+6m$ arcs ce qui se fait en temps polynomial. Nous savons d'après la question 1 que le problème [RS] est dans la classe NP et par hypothèse [3-SAT] est NP-complet. D'autre part nous avons montré que la transformation d'une instance de [3-SAT] en instance de [RS] se faisait en temps polynomial. On peut conclure que [RS] est NP-complet par transitivité de la transformation polynomiale puisque pour tout problème $\Pi \in$ NP, $\Pi \propto$ [3-SAT] et [3-SAT] $\propto$ [RS], donc $\Pi \propto$ [RS].