Informatique II – corrigé

Quelques rappels sur les formules booléennes

On se donne un ensemble $\mathcal{V} = \{a, b, c, \ldots, a_1, b_1, c_1, \ldots\}$ *infini dénombrable de variables propositionnelles.*

On construit les formules booléennes à partir des variables de $\mathcal{V}$*, des constantes* 0 *(faux) et* 1 *(vrai), et des connecteurs usuels,* $\neg$ *(non),* $\vee$ *(ou),* $\wedge$ *(et). Les connecteurs* $\vee$ *et* $\wedge$ *peuvent être n-aires. On note* $var(F)$ *l'ensemble des variables apparaissant dans une formule* F*.*

Une affectation est une fonction de $\mathcal{V}$ *dans* $\{0, 1\}$*. On étend la notion d'affectation aux formules en utilisant les tables de vérités. Une affectation* σ *satisfait la formule* F *si* $\sigma(F) = 1$ *; on dit que* σ *est une solution de* F*. On dit que* σ *falsifie* F *si* $\sigma(F) = 0$*. Une formule* F *est satisfiable s'il existe une affectation* σ *telle que* $\sigma(F) = 1$*. Elle est insatisfiable sinon. Deux formules* F *et* G *(ne portant pas forcément sur les mêmes ensembles de variables) sont dites équivalentes pour la satisfiabilité si on a (*F *satisfiable)* $\Leftrightarrow$ *(*G *satisfiable).*

1 Formules CNF

Un littéral est soit une variable (littéral positif), soit sa négation (littéral négatif). Par souci de simplicité, on note $\neg p$, l'opposé d'un littéral p (sachant que $\neg\neg p$ et p ont les mêmes tables de vérité). Une clause est une disjonction de littéraux aussi manipulée comme l'ensemble de ses littéraux. Une formule sous forme normale conjonctive (CNF) est une conjonction de clauses aussi manipulée comme l'ensemble de ses clauses. On rappelle que toute formule admet une formule CNF équivalente (ayant les mêmes variables et étant satisfaite par les mêmes affectations). On remarque qu'une CNF contenant la clause vide est insatisfiable (la clause vide étant assimilée à la constante 0).

Une formule est dite k-CNF, $k \geq 1$, si c'est une formule CNF dont toutes les clauses comportent au plus k littéraux. Elle est dite exact-k-CNF si toutes ses clauses comportent exactement k littéraux. Finalement, elle est dite k,j-CNF, $k \geq 1, j \geq 0$ (respectivement exact-k,j-CNF) si c'est une formule k-CNF (respectivement exact-k-CNF) et si aucune de ses clauses ne comporte plus de j littéraux positifs. Par convention, on définit la taille $|C|$ d'une clause C comme son nombre de littéraux et la taille $|F|$ d'une formule CNF F comme la somme de la taille de ses clauses.

Soient deux clauses C et D. On dira que la résolvante R de C et D existe, si d'une part il existe un littéral p et des clauses C' et D' tels que $C = \{p\} \cup C'$ et $D = \{\neg p\} \cup D'$, d'autre part aucun autre littéral n'apparait sous des formes opposées dans C' et D'. Si R existe, $R = C' \cup D'$.

Le problème SAT consiste à déterminer si un ensemble de clauses est satisfiable. On rappelle que le problème SAT est NP-complet.

Question 1 :

- *En introduisant une nouvelle variable, donner une formule exact-3-SAT équivalente pour la satisfiabilité à la formule $(p \Leftrightarrow \neg q)$.*
- *Soient F et G deux formules CNF et C et D deux clauses telles que $F = \{C, D\} \cup G$ et soit R la résolvante de C et D (on suppose qu'elle existe). Montrer que F est satisfiable si et seulement si $F \cup \{R\}$ est satisfiable.*

Soit G est une formule quelconque et soit r est une variable n'apparaissant pas dans G, alors les formules G et $F = (G \vee r) \wedge (G \vee \neg r)$ sont équivalentes pour la satisfiabilité. En effet, soit σ une affectation telle que $\sigma(G) = 1$, alors $\sigma(G \vee r) = 1$ et $\sigma(G \vee \neg r) = 1$ quelque soit la valeur de $\sigma(r)$ et donc $\sigma(F) = 1$. Réciproquement soit σ une affectation telle que $\sigma(F) = 1$. Si $\sigma(r) = 0$ alors $\sigma(G) = 1$ puisque σ doit satisfaire $(G \vee r)$. De même, si $\sigma(r) = 1$ alors $\sigma(G) = 1$ puisque σ doit satisfaire de $(G \vee \neg r)$. Donc σ satisfait G.

Par définition du connecteur $\Leftrightarrow$, $p \Leftrightarrow \neg q$ est équivalente à $(p \vee q) \wedge (\neg p \vee \neg q)$. En appliquant la propriété ci-dessus à $(p \vee q)$ d'une part et $(\neg p \vee \neg q)$ d'autre part, on obtient l'ensemble de clauses suivant (qui est donc équivalent pour la satisfiabilité à $(p \Leftrightarrow \neg q)$).

$$\begin{array}{ll} & p \vee q \vee r \\ \wedge & p \vee q \vee \neg r \\ \wedge & \neg p \vee \neg q \vee r \\ \wedge & \neg p \vee \neg q \vee \neg r \end{array}$$

Soit σ une affectation telle que $\sigma(F \cup \{R\}) = 1$, alors $\sigma(F) = 1$. Réciproquement, soit σ une affectation telle que $\sigma(F) = 1$. Alors $\sigma(C) = 1$ et $\sigma(D) = 1$. Posons $C = C' \vee r$ et

$D = D' \vee \neg r$ (et donc $R = C' \vee D'$). Si $\sigma(r) = 0$ alors $\sigma(C') = 1$ et donc $\sigma(R) = 1$. De même, si $\sigma(r) = 1$ alors $\sigma(D') = 1$ et donc $\sigma(R) = 1$. Donc, $\sigma(F \cup \{R\}) = 1$.

Question 2 : *Montrer que tout ensemble de clauses peut être récrit en une formule exact-3-SAT équivalente pour la satisfiabilité.*

Soit F un ensemble de clauses et soit C une clause de F. Faisons une étude de cas sur la taille de C.

- Si $C = \{p\}$, où p est un littéral quelconque, alors on se donne deux variables q et r n'apparaissant pas dans F et en vertu de ce qui précède, F est équivalente pour la satisfiabilité à la formule suivante :

$$(F \setminus C) \cup \{\{p, q, r\}, \{p, q, \neg r\}, \{p, \neg q, r\}, \{p, \neg q, \neg r\}\}$$

- Si $C = \{p, q\}$ où p et q sont deux littéraux quelconques, alors on se donne une variable r n'apparaissant pas dans F et F est équivalente pour la satisfiabilité à la formule suivante :

$$(F \setminus C) \cup \{\{p, q, r\}, \{p, q, \neg r\}\}$$

- Si $|C| = 3$, on garde C telle quelle.
- Si $C = \{p_1, \ldots, p_n\}$, $n > 3$. Alors on se donne les variables r_1, r_2, r_{n-2} n'apparaissant pas dans F, et F est équivalente pour la satisfiabilité à la formule suivante :

$$(F \setminus C) \cup \{\{p_1, p_2, r_1\}, \{\neg r_1, p_3, r_2\} \ldots \{\neg r_{n-2}, p_{n-1}, p_n\}\}$$

On appliquant ce principe sur toutes les clauses de F, on transforme F en une formule exact-3-SAT équivalente.

Question 3 : *Quelle est la complexité du problème exact-3, 2-SAT ?*

Chaque clause d'une instance exact-3, 2-SAT contient au moins un littéral négatif. Par conséquent, l'affectation donnant la valeur 0 à toutes les variables satisfait toutes les clauses. Une fois que l'on sait que l'instance est effectivement exact-3, 2-SAT, sa satisfiabilité est donc décidée en temps constant.

D'autre part, il est facile de tester en tant linéaire si une instance est exact-3, 2-SAT.

Question 4 : *Soient n et k deux entiers strictement positifs, $n > k$, dénombrer le nombre de clauses de taille inférieure ou égale à k que l'on peut construire sur un ensemble de n variables.*

Pour constuire une clause de taille k dans un ensemble de n variables, il faut 1) choisir k variables parmi les n (soit $\binom{n}{k}$ choix) et 2) choisir un signe pour chaque variable (soit 2^k choix). Le nombre recherché $\#Clauses(n, k)$ est donc :

$$\#Clauses(n, k) = \sum_{i=0}^{k} 2^k \binom{n}{k} \tag{1}$$

On note $\setminus$ la différence ensembliste. On considère l'algorithme (dit de saturation) d'un ensemble F de clauses suivant :

SATURATION :
Donnée *: une formule CNF F*
Résultat *: une formule CNF G*
Variables locales *: C_1, C_2, D, R : clauses ; FINI : booléen.*
début *:*
  $G \leftarrow F$
  FINI $\leftarrow$ *faux*
  tant que *non FINI* **faire**
    FINI $\leftarrow$ *vrai*
    pour tous *les couples de clauses C_1, C_2 de G* **faire**
      si *la résolvante R de C_1 et C_2 existe*
      et si *il n'existe pas de clause D de G telle que $D \subseteq R$*
      alors faire
        pour toutes *les clauses D telles que $R \subset D$* **faire**
          $G \leftarrow G \setminus \{D\}$
        fait
        $G \leftarrow G \cup \{R\}$
        FINI $\leftarrow$ *faux*
      fait
    fait
  fait
  retourner G
fin

Remarque : $R \subset D$ si $R \subseteq D$ et $R \neq D$.

Question 5 :
- *Prouver que l'algorithme SATURATION termine pour toute formule CNF F.*
- *Prouver que la formule G produite par l'algorithme SATURATION est équivalente pour la satisfiabilité à la formule F de départ.*

Lorsqu'une résolvante R est ajoutée à G, l'algorithme garantit d'une part qu'il n'existe pas de clause $D \subseteq R$ dans G d'autre part que toutes les clauses D telles que $R \subset D$ sont retirées de G. En conséquence, une telle résolvante R ne peut pas être ajoutée plus d'une fois à l'ensemble. On remarque de plus que toutes les clauses ajoutées à G sont construites sur l'ensemble de variables de départ. Comme on ne peut construire qu'un nombre fini de clauses sur cet ensemble (qui est fini), l'algorithme s'arrête en un temps fini.

On a vu précèdemment que si F est un ensemble de clauses et R est la résolvante de deux clauses de F, alors F est équivalent pour la satisfiabilité à $F \cup \{R\}$.

D'autre part, si F contient deux clauses C et D telles que $C \subset D$, alors toute affectation satisfaisant $F \setminus D$ satisfait C, donc D, donc F.

L'ensemble de clauses produit par l'algorithme SATURATION est donc équivalent pour la satisfiabilité à l'ensemble de départ.

Question 6 : *En supposant que les clauses sont codées par des listes de littéraux et les ensembles de clauses par des listes de clauses, analyser la complexité de la boucle interne de l'algorithme SATURATION (la boucle « pour tous les couples ... »).*

On va analyser la complexité de la boucle interne sans chercher à optimiser les différentes opérations.

On appelle n le nombre de clauses de G. Il y a $n \times (n-1)/2$ couples de clauses. On appelle k la taille maximum des clauses de G. k est majoré par le nombre v de variables apparaissant dans G.

On remarque tout d'abord que tester si un littéral ou son opposé apparaît dans une clause est en $\mathcal{O}(k)$. Pour tester si la résolvante de deux clauses C_1 et C_2 existent, il faut tester qu'un et un seul littéral de C_1 apparait sous la forme opposée dans C_2. On a donc besoin d'un boucle qui teste pour chaque littéral p de C_1 si $p \in C_2$ ou si $\neg p \in C_2$. La complexité de ce test est donc en $\mathcal{O}(k^2)$.

Tester si il existe une clause D de G telle que $D \subseteq R$, demande une itération sur les clauses de G. Chaque test d'inclusion demande à nouveau une double boucle pour vérifier que chaque littéral de D est dans R. La complexité de la recherche d'une clause D de G telle que $D \subseteq R$ est donc $\mathcal{O}(n \times k^2)$.

Pour les mêmes raisons, retirer toutes les clauses telles que $R \subset D$, est en $\mathcal{O}(n \times k^2)$.

Par conséquent la complexité de la boucle interne de l'algorithme SATURATION est en $\mathcal{O}(n^2 \times (k^2 + n \times k^2 + n \times k^2)) = \mathcal{O}(n^3 \times k^2)$.

Question 7 : *En s'aidant des réponses aux questions précédentes, analyser la complexité de l'algorithme SATURATION lorsqu'on l'applique sur des formules 2-SAT. Qu'en conclure sur la complexité du problème 2-SAT ?*

On remarque tout d'abord que la résolvante de deux clauses de taille 2 est au plus de taille 2. D'après la question 4, le nombre de clauses de taille au plus 2 est en $\mathcal{O}(n+2^2 \times n^2) = \mathcal{O}(n^2)$. Il suit que l'algorithme SATURATION fera au plus $\mathcal{O}(n^2)$ itérations de la boucle principal. Par conséquent, il est en $\mathcal{O}(n^2 \times n^3 \times 2^2) = \mathcal{O}(n^5)$ et donc polynomial sur les instances 2-SAT.

Pour en déduire la polynomialité du problème 2-SAT, il reste à prouver que l'algorithme SATURATION est complet sur les instances 2-SAT, c'est à dire qu'il produit la clause vide si l'instance considérée est insatisfiable.

Pour cela considérons une instance 2-SAT F saturée et ne contenant pas la clause vide et supposons que F est insatisfiable.

On va construire de proche en proche une affection σ satisfaisant F, montrant par là une contradiction avec nos hypothèses.

On choisit donc un littéral quelconque p apparaissant dans une clause de F. On pose $\sigma(p) = 1$. Pour toutes les clauses de F de la forme $C = \{\neg p, q\}$, on pose $\sigma(q) = 1$. σ satisfait donc toutes les clauses contenant p et toutes les clauses contenant $\neg p$. Supposons que F contient deux clauses $C = \{\neg p, q\}$ et $D = \{\neg q, r\}$. F étant saturée, il contient aussi la résolvante de C et de D, c'est à dire $\{\neg p, r\}$. Et donc par construction $\sigma(r) = 1$ et donc $\sigma(D) = 1$. Il suit que pour toutes les clauses C contenant un littéral q tel que $\sigma(q) = 0$, on a $\sigma(C) = 1$.

On recommence cette construction tant que toutes les variables de F n'ont pas reçu de valeur.

On construit ainsi une affectation qui satisfait toutes les clauses de F et donc F, ce qui est en contradiction avec notre hypothèse.

On considère l'algorithme (dit de propagation unitaire) dans un ensemble de clauses F suivant :

PROPAGATION-UNITAIRE
Donnée *: une formule CNF F*
Résultat *: une formule CNF G*
Variables locales *: C, D : clauses ; p, q : littéraux ; L : ensemble de littéraux ; FINI : booléen.*
début
  $G \leftarrow F$
  $L \leftarrow \emptyset$
  FINI $\leftarrow$ *faux*
  tant que *non FINI et que G ne contient pas de clause vide* **faire**
    FINI $\leftarrow$ *vrai*
    si *G contient une clause $C = \{p\}$ telle que $p \notin L$* **alors faire**
      $L \leftarrow L \cup \{p\}$
      FINI $\leftarrow$ *faux*
      pour toutes *les clauses D de G, $D \neq C$, telles que $p \in D$* **faire**
        $G \leftarrow G \setminus \{D\}$
      fait
      pour toutes *les clauses D de G, telles que $\neg p \in D$* **faire**
        $D \leftarrow D \setminus \{\neg p\}$
      fait
    fait
  fait
fin

Question 8 : *Analyser la complexité de l'algorithme PROPAGATION-UNITAIRE.*

On appelle v le nombre de variables et c le nombre de clauses de F.

On remarque tout d'abord que la boucle externe de l'algorithme est appellée au plus n fois.

Rechercher une clause unaire est en $\mathcal{O}(c)$. Supprimer ou « raccourcir » toutes les clauses contenant un littéral donné est aussi en $\mathcal{O}(c)$.

L'algorithme est donc en $\mathcal{O}(n \times c)$.

On admettra qu'en choisissant les structures de données adéquates, on peut mettre en œuvre un algorithme de propagation unitaire dont la complexité est en $\mathcal{O}(|F|)$.

Question 9 : *Soient F une formule 2-CNF, p un littéral et G la formule obtenue à partir de $F \cup \{p\}$ par propagation unitaire.*

- *Que peut-on dire de la satisfiabilité de F et G dans le cas où G ne contient pas la clause vide (pour cela, on étudiera ce qu'il advient des clauses de la forme $\{q, x\}$ et $\{\neg q, x\}$, où q est un des littéraux « propagés » par l'algorithme) ?*

– *En utilisant le fait que F est satisfiable si et seulement si $F \cup \{p\}$ est satisfiable ou bien si $F \cup \{\neg p\}$ est satisfiable, en déduire un algorithme de complexité linéaire (en temps) pour tester la satisfiabilité des formules 2-SAT.*

Soit q un des littéraux propagés par l'algorithme. Les clauses de la forme $\{q, r\}$ sont retirées de F. Toute clause de la forme $\{\neg q, r\}$ est remplacée par la clause $\{r\}$ et r est propagé à son tour (par résolution). Il suit que l'ensemble de clauses produit par la propagation unitaire est équivalent pour la satisfiabilité à l'ensemble de départ.

On considère l'algorithme EXPLORATION-UNITAIRE suivant :

EXPLORATION-UNITAIRE
Donnée : une formule CNF F
Résultat : booléen (satisfiabilité de F)
Variable locale : p : variable, G : CNF
début
  si F est vide **alors retourner** vrai
  choisir une variable p apparaissant dans F
  $G \leftarrow$ PROPAGATION-UNITAIRE$(F \cup \{p\})$
  si G contient la clause vide **alors**
    $G \leftarrow$ PROPAGATION-UNITAIRE$(F \cup \{\neg p\})$
  retourner EXPLORATION-UNITAIRE G
fin

Le choix de la variable p, tester si F est vide ou contient la clause vide peut être fait en $\mathcal{O}(1)$ (lors de la propagation unitaire). Dans le pire cas, l'algorithme EXPLORATION-UNITAIRE va faire l'appel à PROPAGATION-UNITAIRE sur $F \cup \{p\}$, trouver la clause vide, faire l'appel à PROPAGATION-UNITAIRE sur $F \cup \{\neg p\}$, puis s'appeler récursivement. La complexité des deux appels à PROPAGATION-UNITAIRE est en $\mathcal{O}(|F|)$. La complexité de EXPLORATION-UNITAIRE est donc en $\mathcal{O}(n \times |F|)$, où n est le nombre de variables.

Remarque : proposer un algorithme équivalent à EXPLORATION-UNITAIRE et analyser sa complexité suffisait à obtenir le maximum de points à cette question.

Pour obtenir une complexité linéaire, il faut remarquer que le problème vient de ce que l'algorithme peut se « tromper » de littéral. L'idée est donc d'effectuer les deux propagations unitaires en parallèle (en faisant par exemple une opération élémentaire à tour de rôle). Si la première des deux qui finit produit la clause vide, on continue l'autre, sinon on abandonne l'autre. On garantit ainsi que l'algorithme élimine paquets de clauses par paquets de clauses en temps linéaire par rapport à la taille des paquets. On en déduit sa linéarité.

Question 10 : *Démontrer que la propagation unitaire est complète sur les $k, 1$-CNF (pour tout $k \geq 1$), c'est à dire qu'une formule $k, 1$-CNF est insatisfiable si et seulement si l'algorithme PROPAGATION-UNITAIRE produit la clause vide.*

$\Leftarrow$: la propagation unitaire ne fait que des résolutions. La formule produite est donc équivalente pour la satisfiabilité à la formule de départ. En particuler, si cette dernière contient la clause vide, c'est que la formule de départ est insatisfiable.
$\Rightarrow$: Soit F la formule obtenue après propagation unitaire. Supposons que F ne contient pas la clause vide. Si F contient une clause unaire $\{p\}$, où p est une variable, alors F ne contient pas de clauses contenant $\neg p$ (puisque ces clauses sont éliminées par la propagation unitaire). Toutes les autres clauses contiennent donc au moins un littéral négatif. On consière l'affectation σ qui donne la valeur 1 à toutes les variables p apparaissant dans une clause de unaire $\{p\}$ et la valeur 0 à toutes les autres variables. Cette affectation satisfait F et donc la formule de départ.
QED

2 Systèmes d'équations booléennes

On considère maintenant des systèmes (c'est à dire des ensembles) d'équations booléennes, construits à partir de l'ensemble $\mathcal{V}$ des variables propositionnelles des connecteurs $\neg$, $\vee$ et $\wedge$ et d'un ensemble infini dénombrable de noms de formules $\mathcal{G} = \{G_1, G_2, \ldots\}$ de la façon suivante : l'ensemble des systèmes d'équations booléennes est le plus petit ensemble tel que :

- *L'ensemble vide $\emptyset$ est un système d'équations booléennes.*
- *Si S est un système d'équations booléennes, $f_1, \ldots, f_n$ sont des noms de formules apparaissant dans S ou des variables propositionnelles et si G_i est un nom de formule n'apparaissant pas dans S alors les trois ensembles suivants sont des systèmes d'équations booléennes :*
  - $S \cup \{G_i = \neg f_1\}$,
  - $S \cup \{G_i = f_1 \vee \ldots \vee f_n\}$,
  - $S \cup \{G_i = f_1 \wedge \ldots \wedge f_n\}$.

Un système d'équations booléennes peut être représenté par un graphe orienté acyclique (DAG) dont les nœuds sont étiquetés par des variables, des noms de formules ou des connecteurs, chaque variable ou nom de formule n'étiquetant qu'un seul nœud. Par exemple, le système d'équations booléennes suivant S_1 :

$$\begin{aligned} G1 &= G2 \vee G3 \\ G2 &= e1 \wedge G4 \\ G3 &= G4 \wedge e3 \\ G4 &= \neg\, e2 \end{aligned}$$

est représenté par le DAG D_1 (Figure 1).

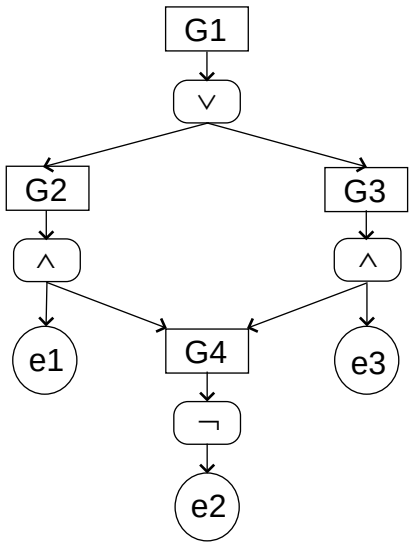

FIG. 1 – Le DAG D_1 représentant S_1.

On associe de façon naturelle une formule booléenne à chaque nœud du DAG étiqueté par un nom de formule. Par exemple, dans le DAG D_1, on associe :

– $\neg e_2$ *au nœud étiqueté par* G_4,
– $(e_1 \wedge \neg e_2)$, *au nœud étiqueté par* G_2,
– $(\neg e_2 \wedge e_3)$ *au nœud étiqueté par* G_3,
– *et* $(e_1 \wedge \neg e_2) \vee (\neg e_2 \wedge e_3)$ *au nœud étiqueté par* G_1.

On suppose que l'on dispose du type de données « DAG ». Ce type de données permet en particulier d'accéder à la liste des nœud fils et à la liste des nœuds pères d'un nœud, ainsi que de marquer les nœuds lors d'un parcours.

Question 11 : *Écrire un algorithme polynomial en temps prenant en entrées un DAG D et un nœud n de D et produisant en sortie la liste des variables étiquetant les nœuds descendants de n (c'est à dire les nœuds du sous-DAG dont n est la racine). Donner la complexité de cet algorithme.*

On commence par écrire un algorithme permettant de marquer les descendants d'un nœud n avec une valeur v. On suppose que si un descendant m de n est marqué avec v, alors tous ses descendants le sont. D'une manière générale, on va supposer dans ce qui suit que tous les nœuds du DAG sont marqués 0. Les algorithmes conserveront cet invariant.

MARQUER-DESCENDANTS
Donnée : n : nœud, v : entier
Variable locale : m : nœud
début
  si n est marqué v **alors retourner**
  marquer n avec v
  pour tout fils m de n **faire**
    MARQUER-DESCENDANTS(m,v)
  fait
fin

Soit F le sous-graphe de racine n. L'algorithme MARQUER-DESCENDANTS passe une et une seule fois par chaque arête de F. Par conséquent, il est en $\mathcal{O}(|F|)$.

Sur le même principe, on écrit une procédure qui collecte les variables non marquées dans un sous-graphe.

COLLECTER-VARIABLES-NON-MARQUÉES
Donnée : n : nœud, L : liste
Variable locale : m : nœud
début
  si n est marqué 1 **alors retourner**
  si n est étiqueté par une variable v **alors** $L \leftarrow L \cup \{v\}$
  marquer n avec 1
  pour tout fils m de n **faire**
    COLLECTER-VARIABLES-NON-MARQUÉES(m,L)
  fait
fin

L'ajout en début (ou fin) de liste est en $\mathcal{O}(1)$, par conséquent COLLECTER-VARIABLES-NON-MARQUÉES est en $\mathcal{O}(|F|)$, où F est le sous-graphe de racine n.

L'algorithme recherché est donc le suivant :

COLLECTER-VARIABLES
Donnée : n : nœud
Résultat : L : liste
début
  $L \leftarrow \emptyset$
  COLLECTER-VARIABLES-NON-MARQUÉES(n,L)
  MARQUER-DESCENDANTS(n,0)
fin

COLLECTER-VARIABLES est évidemment aussi en $\mathcal{O}(|F|)$.

On définit le poids d'un nœud de la façon suivante :
– 1 si c'est une racine (i.e. s'il n'a pas de nœud père),
– la somme des poids de ses nœuds pères sinon.

Question 12 : *Écrire un algorithme polynomial en temps prenant en entrée un DAG D et calculant le poids de chaque nœud de D. Donner sa complexité.*

L'algorithme se décompose en deux étapes : on va commencer par créer une liste de tous les nœuds du graphe ordonnée de façon à ce qu'un père soit toujours après ses fils. On se servira ensuite de cette liste pour calculer les poids.

COLLECTER-NŒUDS-NON-MARQUÉS
Donnée : n : nœud L : liste
Variable locale : m : nœud
début
  si n est marqué 1 **alors retourner**
  ajouter n en queue dans L
  marquer n avec 1
  pour tout fils m de n **faire**
    COLLECTER-NŒUDS-NON-MARQUÉS(m,L)
  fait
fin

COLLECTER-NŒUDS-NON-MARQUÉS est en $\mathcal{O}(|F|)$, où F est le sous-graphe de racine n, pour les mêmes raisons que précédemment.

COLLECTER-NŒUDS
Donnée : F : DAG
Résultat : L : liste
Variable locale : n : nœud
début
  $L \leftarrow \emptyset$
  pour tout nœud n de F **faire**
    COLLECTER-NŒUDS-NON-MARQUÉS(n,L)
  fait
  pour tout nœud n de F **faire**
    marquer n avec 0
  fait
fin

On remarque que cet algorithme traverse, lui-aussi une et une seule fois chaque arête du graphe. Par conséquent, il est en $\mathcal{O}(|F|)$.

On peut maintenant calculer les poids.

CALCULER-POIDS
Donnée : F : DAG
Variable locale : L : liste, n, m : nœud, p : entier
début
  $L \leftarrow$ COLLECTER-NŒUDS(F)
  pour tout nœud n de L **faire**
    $p \leftarrow 0$
    pour tout fils m de n **faire**
      $p \leftarrow p + \text{poids}(m)$
    fait
    $\text{poids}(n) \leftarrow p$
  fait
fin

Le calcul des poids passe lui aussi une et une seule fois par chaque arête du graphe. Par conséquent CALCULER-POIDS est en $\mathcal{O}(|F|)$.

On suppose à partir de maintenant et dans toute la suite du sujet que :
- *Les connecteurs $\vee$ et $\wedge$ sont binaires.*
- *Tous les DAG considérés n'ont qu'une seule racine.*

La polarité $pol(v)$ d'une variable v dans un DAG avec une seule racine est définie de la façon suivante :
- *$pol(v)$ vaut 1 si tous les chemins de la racine au nœud étiqueté par v contiennent un nombre pair (incluant 0) de nœuds étiquetés par des négations.*

- *$pol(v)$ vaut -1 si tous les chemins de la racine au nœud étiqueté par v contiennent un nombre impair de nœuds étiquetés par des négations.*
- *$pol(v)$ vaut 0 dans les autres cas.*

Question 13 : *Écrire un algorithme polynomial en temps prenant en entrée un DAG D et calculant la polarité de chaque nœud de D étiqueté par une variable. Donner la complexité de cet algorithme.*

On suppose que les polarités ont été initialisées à la valeur 2 (interprétée par « inconnue »). On écrit un algorithme récursif travaillant de haut en bas à partir des racines du DAG.

CALCULER-POLARITÉ
Donnée : n nœud, p : entier
Variable locale : m : nœud, q : entier
début
  si $pol(n) = p$ **alors retourner**
  si $pol(n) = 2$ **alors** $q \leftarrow p$ **sinon** $q \leftarrow 0$
  $pol(n) \leftarrow q$
  si $type(n) = \neg$ **alors** $q \leftarrow -q$
  pour tout fils m de n **faire**
    CALCULER-POLARITÉ(m,q)
  fait
fin

On parcours maintenant tous les nœuds du graphes pour initialiser les polarités et appeller l'algorithme ci-dessus.

CALCULER-POLARITÉS
Donnée : F : DAG
Variable locale : n : nœud
début
  pour tout nœud n de F **faire**
    $pol(n) \leftarrow 2$
  fait
  pour tout nœud n de F **faire**
    si n n'a pas de père **alors** CALCULER-POLARITÉ(m,1)
  fait
fin

La polarité des nœuds est initialisée en $\mathcal{O}(|F|)$. Chaque arête du DAG est ensuite traversée au plus deux fois : une fois pour passer de 2 à 1 ou -1, et une autre fois pour passer de cette valeur à 0. Par conséquence l'algorithme CALCULER-POLARITÉS est en $\mathcal{O}(|F|)$.

Un système d'équations booléennes est sous forme normale négative (NNF) si les négations n'apparaissent que sur les nœuds pères des nœuds étiquetés par des variables. On rappelle les lois de de Morgan :

$$\begin{aligned} \neg(F \vee G) &\equiv \neg F \wedge \neg G \\ \neg(F \wedge G) &\equiv \neg F \vee \neg G \end{aligned}$$

Question 14 : *En utilisant les lois de de Morgan, écrire un algorithme polynomial en temps, dont on précisera la complexité, prenant un DAG D en entrée et produisant un DAG D' en sortie tel que :*

- *D' n'a qu'une seule racine,*
- *D' est sous forme normale négative,*
- *la formule associée à la racine de D' est équivalente (au sens habituel des tables de vérité) à la formule associée à la racine de D.*

Idée : On pourra introduire de nouveaux noms de formules.

L'idée est la suivante : on crée pour chaque nœud n du DAG deux nouveaux : un nœud $nnfpos(n)$ pour coder (la formule codée par) n sous forme NFF et un nœud $nnfneg(n)$ pour coder la négation sous forme NFF de n. Pour les variables, on aura nnfpos(n)=n. Ces deux nœuds sont créés à partir des nœuds nnfpos et nnfneg des fils de n. On doit donc considérer les nœuds du DAG de bas en haut. Pour cela on réutilise la procédure COLLECTER-NŒUDS de la question 12. On suppose définies les procédures CRÉER-NON, CRÉER-OU et CRÉER-ET qui créent des nœuds étiquetés par ces connecteurs.

CRÉER-NNF
Donnée : F : DAG
Variable locale : L : liste, $n, m, m_1, \ldots, m_k$: nœud
début
  $L \leftarrow$ COLLECTER-NŒUDS(F)
  pour tout nœud n de L **faire**
    si n est une variable terminale (une feuille) **alors**
      $nnfpos(n) \leftarrow n$
      $nnfneg(n) \leftarrow$ CRÉER-NON(n)
    sinon si n est une variable non terminale **alors**
      $nnfpos(n) \leftarrow n$
      $nnfneg(n) \leftarrow nnfneg(fils(n))$
    sinon si $n =$ NON(m) **alors**
      $nnfpos(n) \leftarrow nnfneg(m)$
      $nnfneg(n) \leftarrow nnfpos(m)$
    sinon si $n =$ OU(m_1,..., m_k) **alors**
      $nnfpos(n) \leftarrow$ CRÉER-OU($nnfpos(m_1)$,...,$nnfpos(m_k)$)
      $nnfneg(n) \leftarrow$ CRÉER-ET($nnfneg(m_1)$,..., $nnfneg(m_k)$)
    sinon si $n =$ ET(m_1,..., m_k) **alors**

$nnfpos(n) \leftarrow$ CRÉER-ET($nnfpos(m_1)$,..., $nnfpos(m_k)$)
$nnfneg(n) \leftarrow$ CRÉER-OU($nnfneg(m_1)$,..., $nnfneg(m_k)$)
fait
fin

Les seules négations créés par CRÉER-NFF sont bien au dessus de variables terminales. D'autre part, la construction suivant les lois de de Morgan, les nœuds $nnfpos$ et $nnfneg$ codent bien respectivement la formule codée par n et sa négation.

D'autre part, chaque nœud du graphe n'est traité qu'une fois. CRÉER-NNF est donc en $\mathcal{O}(|F|)$.

Question 15 : *Écrire un algorithme polynomial en temps prenant en entrée un DAG D et produisant en sortie une formule CNF F telle que F est équivalente pour la satisfiabilité à la formule associée à la racine de D. Donner la complexité de cet algorithme.*

On remarque que chaque nœud du DAG définit en fait une équivalence : par exemple un nœud $n : OU(n_1, n_2)$ définit l'équivalence $n \Leftrightarrow n_1 \vee n_2$. Le DAG définit donc la conjonction de ces équivalences. On peut mettre chaque équivalence sous forme d'un ensemble de clauses :

- $n \Leftrightarrow \neg m$ donne l'ensemble de clauses $(n \vee m) \wedge (\neg n \vee \neg m)$.
- $n \Leftrightarrow (m_1 \vee m_2)$ donne l'ensemble de clauses $(n \vee \neg m_1) \wedge (n \vee \neg m_2) \wedge (\neg n \vee m_1 \vee m_2)$.
- $n \Leftrightarrow (m_1 \wedge m_2)$ donne l'ensemble de clauses $(n \vee \neg m_1 \vee \neg m_2) \wedge (\neg n \vee m_2) \wedge (\neg n \vee m_2)$.

Pour obtenir la formule codée par la racine r du DAG, il suffit donc d'ajouter à l'ensemble de clauses, la clause unaire $\{r\}$.

Chaque nœud du graphe n'étant traité qu'une fois et la compilation de chaque nœud pouvant se faire en temps constant l'algorithme ci-dessus est donc en $\mathcal{O}(|F|)$.