1 Problèmes indécidables sur les grammaires non contextuelles

1.1 Le Problème de Correspondance de Post

1. Cette instance de PCP est positive ; voici une solution :

$$i_1 = 3\ ; \quad i_2 = 2\ ; \quad i_3 = 3\ ; \quad i_4 = 1$$

$$\begin{pmatrix} bba \\ bb \end{pmatrix} \cdot \begin{pmatrix} ab \\ aa \end{pmatrix} \cdot \begin{pmatrix} bba \\ bb \end{pmatrix} \cdot \begin{pmatrix} a \\ baa \end{pmatrix} = \begin{pmatrix} bbaabbbaa \\ bbaabbbaa \end{pmatrix}$$

2. Cette instance ne possède pas de solution, car pour tout domino, on a $|u_i| > |v_i|$, donc peu importe la suite de dominos considérée, le mot du haut sera toujours strictement plus long que le mot du bas.
3. Si une instance de PCP est positive, soit $d_{i_1} \cdots d_{i_k}$ une suite de dominos constituant une solution. Alors, $\forall n \in \mathbb{N}^\star$, $(d_{i_1} \cdots d_{i_k})^n$ est également une solution. Il y a donc bien une infinité de solutions.

1.2 Semi-décidabilité de PCP

4.
```c
int longueur(char* s) {
  int i = 0;
  while (s[i] != '\0') {
    i++;
  }
  return i;
}
```

5.
```c
bool egales(char* s1, char* s2) {
  int i = 0;
  while (s1[i] == s2[i]) {
    if (s1[i] == '\0') {
      return true;
    }
    i++;
  }
  return false;
}
```

6.
```c
bool prefixe(char* p, char* s) {
  for (int i = 0; p[i] != '\0'; i++) {
    if (s[i] == '\0') { return false; } // s est plus petite que p
    if (p[i] != s[i]) { return false; } // mauvais caractère
  }
  return true;
}
```

7.
```c
char* concat(char* s1, char* s2) {
  int n1 = longueur(s1);
  int n2 = longueur(s2);
  char* res = malloc((n1+n2+1) * sizeof(char));
  for (int i = 0; i < n1; i++) {
    res[i] = s1[i];
  }
  for (int i = 0; i <= n2; i++) { // n2 inclus pour avoir '\0' à la fin
    res[n1+i] = s2[i];
  }
  return res;
}
```

8. Si on considère que K est écrit en unaire sur l'entrée, alors un tel certificat est bien de taille polynomiale en la taille de l'entrée, et dans ce cas, on peut dire que BPCP $\in$ NP. Mais si on considère que K est écrit sur l'entrée en binaire (ou en base 10, ou en n'importe quelle base ≥ 2), ce qui est plus usuel, alors le certificat considéré n'est pas de taille polynomiale en la taille de K (qui est un $\mathcal{O}(\log(K))$).
Remarque : une variante qui est bien dans NP même si K n'est pas écrit en unaire est la suivante (elle est même NP-complète) :

BPCP	
Entrée :	une liste $\left[\binom{u_1}{v_1}, \dots, \binom{u_n}{v_n}\right]$ de dominos, un entier $K \leq n$;
Question :	$\exists k \leq K,\ \exists (i_1, \dots, i_k) \in [\![1, n]\!]^k,\ u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$?

9.
```c
bool bpcp_backtrack(char** u, char** v, char* haut, char* bas, int n, int k) {
  if (k == 0) { return false; }
  for (int i = 0; i < n; i++) {
    char* nouveau_haut = concat(haut, u[i]);
    char* nouveau_bas = concat(bas, v[i]);
    bool trouve = egales(nouveau_haut, nouveau_bas) || // évaluation paresseuse
      ((prefixe(nouveau_haut, nouveau_bas) || prefixe(nouveau_bas, nouveau_haut))
       && bpcp_backtrack(u, v, nouveau_haut, nouveau_bas, n, k-1));
    free(nouveau_haut);
    free(nouveau_bas);
    if (trouve) { return true; }
  }
  return false;
}

bool bpcp(char** u, char** v, int n, int K) {
  return bpcp_backtrack(u, v, "", "", n, K);
}
```

10.
```c
bool pcp(char** u, char** v, int n) {
  int k = 1;
  while (!bpcp(u, v, n, k)) {
    k++;
  }
  return true;
}
```

1.3 Problèmes de décision sur les grammaires non contextuelles

11. Si $f_1 \leq_{\text{DEC}} f_2$, il existe $g : E_1 \to E_2$ calculable telle que $f_1 = f_2 \circ g$. Notons `a_g` une fonction OCaml qui implémente g. Par l'absurde, supposons f_2 décidable, on peut donc implémenter une fonction OCaml `a_f2` qui calcule f_2. On pourrait alors implémenter la fonction suivante qui calcule f_1 :

```
let a_f1 x = a_f2 (a_g x)
```

Ce qui est absurde, car f_1 est indécidable. Donc f_2 est indécidable.

12. Voici une grammaire non contextuelle $\mathcal{G}_u$ reconnaissant L_u :

$$S \to u_1 S d_1 \mid \cdots \mid u_n S d_n$$
$$S \to u_1 d_1 \mid \cdots \mid u_n d_n$$

13. (a) Si $\left[\binom{u_1}{v_1}, \ldots, \binom{u_n}{v_n}\right]$ est une instance positive de PCP, notons $i_1, \ldots, i_k$ une liste d'indices correspondant à une solution. On peut alors générer le mot $u_{i_1} \cdots u_{i_k} d_{i_k} \cdots d_{i_1}$ dans L_u et le mot $v_{i_1} \cdots v_{i_k} d_{i_k} \cdots d_{i_1}$ dans L_v, $u_{i_1} \cdots u_{i_k} d_{i_k} \cdots d_{i_1} = v_{i_1} \cdots v_{i_k} d_{i_k} \cdots d_{i_1}$, donc que $L_u \cap L_v \neq \varnothing$. Réciproquement, si $L_u \cap L_v \neq \varnothing$, considérons $w \in L_u \cap L_v$. Comme $w \in L_u$, on a que w est de la forme : $u_{i_1} \cdots u_{i_k} d_{i_k} \cdots d_{i_1}$, et Comme $w \in L_v$, on a que w est de la forme : $v_{i'_1} \cdots v_{i'_k} d_{i'_k} \cdots d_{i'_1}$, et on a forcément que $\forall j,\ i_j = i'_j$. Ainsi, les indices $i_1, \ldots, i_k$ correspondent à une solution de l'instance de PCP, cette instance est donc positive.

(b) Construire $\mathcal{G}_u$ reconnaissant L_u (et similairement $\mathcal{G}_v$ reconnaissant L_v) peut bien se faire algorithmiquement à partir de l'instance de PCP considérée, et d'après la question précédente, l'instance de PCP est positive si et seulement si les grammaires ainsi obtenues sont une instance négative de InterVide. Ainsi, en passant à la négation (ce qui ne pose pas de problème pour les résultats d'indécidabilité, contrairement aux résultats de NP-difficulté), on a bien PCP $\leq_{\text{DEC}}$ InterVide. Donc InterVide est indécidable.

14. (a) Cette grammaire reconnaît le langage $L_u \cup \mathsf{L}_v$.

(b) La grammaire de la question précédente se calcule bien algorithmiquement à partir de l'instance de PCP considérée. De plus, cette grammaire est ambigüe si et seulement si $L_u \cap L_v = \varnothing$, donc (même raisonnement que pour la question 13b) si et seulement si l'instance de PCP est positive. Donc PCP $\leq_{\text{DEC}}$ Ambig, et Ambig est indécidable.

15. (a) $L'_u \cup \overline{\Sigma^\star \Delta^\star} \cup \{\varepsilon\} = \overline{L_u}$.

(b) Pour qu'un mot soit dans $\overline{\Sigma^\star \Delta^\star}$, il suffit qu'il possède une lettre de Δ avant une lettre de Σ. On construit donc une grammaire qui commence par générer une telle paire de lettres, puis le symbole T permet de générer n'importe quels mots autour de ces lettres. On obtient les règles suivantes :

$$S \to TdTaT \qquad \forall (a,d) \in \Sigma \times \Delta$$
$$T \to cT \qquad \forall c \in \Sigma'$$
$$T \to \varepsilon$$

(c) On commence par des règles similaires à la grammaire reconnaissant L_u, mais sans pouvoir finir de générer un mot (l'idée est qu'on peut commencer à générer un mot de L_u avant d'introduire un problème) :

$$S \to u_1 S d_1 \mid \cdots \mid u_n S d_n$$

Puis, lorsqu'on introduit un problème qui fait qu'on n'est pas dans L_u, il y a 3 cas :

- soit on introduit à droite une lettre d_i, et on introduit à gauche un mot $u \neq u_i$ tel que :
  - soit $|u| = |u_i|$ (ce cas sera géré par le symbole X), et dans ce cas on peut continuer à générer n'importe quelle lettre de Σ à gauche et de Δ à droite ;
  - soit $|u| < |u_i|$ (ce cas sera géré par le symbole Y), et dans ce cas il ne faut plus générer de lettre de Σ à gauche (sinon, on risque de "réparer" le problème), on peut en revanche générer n'importe quelle lettre de Δ à droite ;
  - soit $|u| > |u_i|$ (ce cas sera géré par le symbole Z), et dans ce cas il ne faut plus générer de lettre de Δ à droite (sinon, on risque de "réparer" le problème), on peut en revanche générer n'importe quelle lettre de Σ à gauche. Grâce à cette dernière remarque, il suffit donc de d'abord générer un tel mot u avec $|u| = |u_i + 1|$, puis laisser les autres lettres être générées arbitrairement par une autre règle.
- soit un introduit à gauche un mot u_i et on n'introduit par la lettre d_i correspondante à droite, mais en fait dans ce cas on peut se ramener au dernier sous-cas ci-dessus.

On obtient alors la grammaire $\mathcal{G}'_u$ suivante :

$$
\begin{array}{ll}
S \to u_i S d_i & \forall i \in [\![1, n]\!] \\
S \to u X d_i & \forall i \in [\![1, n]\!],\ \forall u \text{ tel que } |u| = |u_i| \text{ et } u \neq u_i \\
S \to u Y d_i & \forall i \in [\![1, n]\!],\ \forall u \text{ tel que } |u| < |u_i| \\
S \to u_i b Z d_i & \forall i \in [\![1, n]\!],\ \forall b \in \Sigma \\
X \to X d_i & \forall i \in [\![1, n]\!] \\
X \to b X & \forall b \in \Sigma \\
X \to \varepsilon & \\
Y \to Y d_i & \forall i \in [\![1, n]\!] \\
Y \to \varepsilon & \\
Z \to b Z & \forall b \in \Sigma \\
Z \to \varepsilon &
\end{array}
$$

Cette grammaire possède bien un nombre fini de règles, car pour chaque $i \in [\![1, n]\!]$, il y a bien un nombre fini de mots u tels que $|u| \leq |u_i|$.

16. Les grammaires des deux question précédentes sont bien calculables à partir de l'instance de PCP considérée. On les fusionne (avec le même symbole S), et on rajoute la règle :

$$S \to \varepsilon$$

On obtient alors une grammaire non contextuelle qui reconnait $L'_u \cup \overline{\Sigma^\star \Delta^\star} \cup \{\varepsilon\} = \overline{L_u}$.
On construit de manière similaire une grammaire non contextuelle reconnaissant $\overline{L_v}$.
On fusionne toutes les règles de ces deux grammaires pour obtenir une grammaire reconnaissant $\overline{L_u} \cup \overline{L_v}$. On remarque alors que $\overline{L_u} \cup \overline{L_v} = \Sigma'^\star$ si et seulement si $L_u \cap L_v = \varnothing$ si et seulement si l'instance de PCP est positive. On a donc bien PCP $\leq_{\text{DEC}}$ UNIV, donc UNIV est indécidable.

17. On commence par définir la grammaire $\mathcal{G}_0$ suivante qui reconnait $\Sigma'^\star$:

$$
\begin{array}{ll}
S \to bS & \forall b \in \Sigma' \\
S \to \varepsilon &
\end{array}
$$

Ainsi, soit $\mathcal{G}$ une grammaire non contextuelle. $\mathcal{G}$ est une instance positive de UNIV si et seulement si $(\mathcal{G}, \mathcal{G}_0)$ est une instance positive de ÉGAL. Donc UNIV $\leq_{\text{DEC}}$ ÉGAL.
Or UNIV est indécidable, donc ÉGAL est indécidable.

2 Problèmes NP-complets

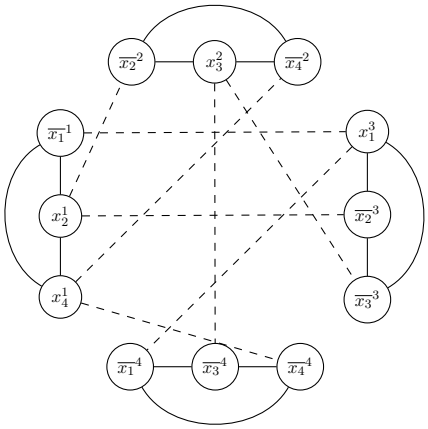
2.1 Couverture de graphe

18. On représente de manière usuelle $S = [\![0, n-1]\!]$ et G par listes d'adjacence, et on choisit comme ensemble de certificats l'ensemble des tableaux de booléens. Un tel tableau `t` de taille n représente alors un ensemble $S' \subseteq S$ ($s \in S' \iff$ `t.(s) = true`, et on a f_{verif} qui doit :
    - vérifier que le tableau est de longueur K ;
    - vérifier que le S' correspondant est bien un ensemble indépendant : on vérifie que chaque paire de sommets de S' n'est pas reliée par une arête de A, ce qui se fait en $\mathcal{O}(|S|^2|A|)$.

    De plus, si un tel certificat existe, il est de taille n, ce qui n'est pas polynomial en la taille de K si K est écrit en binaire, mais c'est bien polynomial en $|S|$ car $K \leq |S|$.
    Ainsi, IndependentSet $\in$ NP.
    Remarque : on aurait pu aussi choisir comme certificats des listes de sommets, et il faudrait alors que la liste soit de taille K pour qu'elle soit valide, et on aurait aussi besoin de la contrainte $K \leq |S|$ pour justifier que le certificat valide est bien de taille polynomiale en la taille de l'entrée. Ici, on aurait pu aussi s'en sortir sans la contrainte $K \leq |S|$ en modifiant f_{verif} pour qu'elle renvoie `false` sans faire aucun autre test si $K > |S|$, car dans ce cas l'instance est forcément négative.

19. On choisit les mêmes certificats, et la seule différence et qu'il faut maintenant vérifier que l'ensemble S' correspondant au tableau est une couverture du graphe, ce qui se fait en $\mathcal{O}(|A| \times |S|)$.
    Donc VertexCover $\in$ NP.

20.



21. Supposons que $\nu \vDash \varphi$. Ainsi, $\forall j \in [\![1, k]\!]$, il existe un littéral $\ell_j \in C_j$ tel que $\nu \vDash \ell_j$.
    On pose $S' = \{\ell_j \mid j \in [\![1, k]\!]\}$. C'est bien un ensemble indépendant de G_φ de taille k car :
    - chaque ℓ_j correspond à une clause différente, donc ils ne sont pas reliés par des arêtes de l'étape 2 ;
    - il n'y a pas non plus d'arête de la forme $\ell_j - \ell_{j'}$ créée à l'étape 3, car sinon on aurait $\overline{\ell_j} = \ell_{j'}$, ce qui est absurde car $\nu \vDash \ell_j$ et $\nu \vDash \ell_{j'}$.

    Réciproquement, supposons que G_φ possède un ensemble indépendant S' de taille k. Grâce aux arêtes créées à l'étape 3, S' ne peut pas posséder deux sommets correspondant respectivement à un ℓ et à $\overline{\ell}$. Ainsi, on peut construire une valuation ν satisfaisant tous les littéraux associés aux sommets de S'.
    De plus, grâce aux arêtes créées à l'étape 2, S' ne peut pas posséder deux sommets correspondant à une même clause, donc comme $|S'| = k$, S' possède exactement un sommet associé à chaque clause. Ainsi, $\forall j \in [\![1, k]\!]$, il existe un littéral $\ell_j \in C_j$ tel que $\nu \vDash \ell_j$, donc $\nu \vDash \varphi$.

22. Le graphe G_φ se construit bien en temps polynomial en $|\varphi|$, car il possède $|\varphi|$ sommets et au plus $|\varphi|^2$ arêtes. Donc, d'après la question précédente, 3-Sat $\leq_{\mathsf{P}}$ IndependentSet.

23. Donc, comme 3-Sat est NP-difficile, IndependentSet est NP-difficile.
    De plus (question 18), IndependentSet $\in$ NP. Donc IndependentSet est NP-complet.

24. Soit $G = (S, A)$ un graphe et $S' \subseteq S$. On montre d'abord que S' est un ensemble indépendant de G si et seulement si $\overline{S'}$ est une couverture de G.

$$\begin{aligned} & S' \text{ est un ensemble indépendant de } G \\ \iff & \forall (u,v) \in S'^2,\ (u,v) \notin A \\ \iff & \forall (u,v) \in A,\ u \notin S' \vee v \notin S' \\ \iff & \forall (u,v) \in A,\ u \in \overline{S'} \vee v \in \overline{S'} \\ \iff & \overline{S'} \text{ est une couverture de } G \end{aligned}$$

Ainsi, (G, K) est une instance positive de IndependentSet si et seulement si $(G, n-K)$ est une instance positive de VertexCover (où n est le nombre de sommets de G). Calculer $(G, n-K)$ à partir de (G, K) se fait bien en temps polynomial, donc IndependentSet $\leq_{\mathsf{P}}$ VertexCover.

25. Donc, comme IndependentSet est NP-difficile, VertexCover est NP-difficile.
De plus (question 19), VertexCover $\in$ NP. Donc VertexCover est NP-complet.

2.2 Complexité grammaticale

2.2.1 Préliminaires

26. Supposons que $\mathcal{G}$ ne soit pas acyclique. Alors il existe $X \in \mathcal{V}$ et $k > 0$ tels que $X \Rightarrow^k \alpha$, avec $|\alpha|_X > 0$. Cela signifie qu'il existe des variables $Y_0 = X, Y_1, \ldots, Y_m = X$ et des règles $Y_i \to \alpha_i Y_{i+1} \beta_i$ pour $i \in [\![0, m-1]\!]$. Ainsi, s'il existait un ordre topologique $(X_0, \ldots, X_{n-1})$ et un indice i tel que $X_i = X$, alors par transitivité de $<$ et par définition de l'ordre topologique, on aurait $i < i$: absurde.

Réciproquement, supposons que $\mathcal{G}$ soit acyclique. Construisons une suite de variables comme suit :

- on pose $Y_0 \in \mathcal{V}$ quelconque ;
- pour $i \in \mathbb{N}$, on pose Y_{i+1} une variable apparaissant dans le membre droite d'une des règles de production depuis Y_i (si un tel y_{i+1} existe.

Le nombre de variables étant fini, si ce processus ne s'arrêtait pas, par principe des tiroirs, on aurait $i < j$ tels que $Y_i = Y_j$, ce qui est impossible car $\mathcal{G}$ est acyclique. Il existe donc une variable X qui ne possède aucune variable dans les parties droites de ses règles de dérivation. Dès lors, on pose $X_{n-1} = X$, et en supprimant toutes les règles concernant X, on obtient à nouveau une grammaire acyclique, donc on peut réitérer ce processus pour continuer de construire un ordre topologique.

27. $\mathcal{G}$ étant acyclique, elle possède un ordre topologique $(X_0, \ldots, X_{n-1})$.
Pour $X \in \mathcal{V}$, on note $\mathcal{L}_X(\mathcal{G}) = \{u \in \Sigma^\star \mid X \Rightarrow^* u\}$.
Montrons par récurrence décroissante que $\forall i \in [\![0, n-1]\!],\ \mathcal{L}_{X_i}(\mathcal{G})$ est fini :

- $\mathcal{L}_{X_{n-1}}(\mathcal{G})$ est fini, X_{n-1} ne peut pas engendrer de variable, donc $|\mathcal{L}_{X_{n-1}}(\mathcal{G})|$ est égal au nombre de règles de la forme $X_{n-1} \to \alpha$;
- supposons le résultat vrai $\forall j \in [\![i, n-1]\!]$ pour un i fixé. Il y a un nombre fini de règles de la forme $X_i \to \alpha$, et les α ainsi générés ne contiennent que des variables X_j avec $j > i$, donc, par (HR), toutes ces variables X_j ne génèrent qu'un nombre fini de mots. Ainsi, $\mathcal{L}_{X_i}(\mathcal{G})$ est fini.

Ainsi, en particulier, $\mathcal{L}(\mathcal{G}) = \mathcal{L}_S(\mathcal{G})$ est fini.

28. (a) Comme pour les graphes, on utilise un système de 3 couleurs pour détecter un cycle. On utilise également une exception pour interrompre la boucle principale dès qu'un cycle est trouvé.

```
type couleur = Blanc | Gris | Noir
exception Cycle

let acyclique g =
  let n = Array.length g in
  let couleurs = Array.make n Blanc in
  let rec existe_cycle i = match couleurs.(i) with
    | Gris -> true
    | Noir -> false
    | Blanc ->
      couleurs.(i) <- Gris ;
      let traiter_symb s = match s with
        | T _ -> false
        | V j -> existe_cycle j
      in
      let traiter_mot alpha = List.exists traiter_symb alpha in
      let b = List.exists traiter_mot g.(i) in
      couleurs.(i) <- Noir ;
      b
  in
  try
    for i = 0 to n-1 do
      if existe_cycle i then raise Cycle
    done ;
    true
  with Cycle -> false
```

(b) La création du tableau `couleurs` se fait en $\mathcal{O}(|\mathcal{V}|)$, et chaque règle de production n'est parcourue qu'une seule fois (et à chaque fois, on doit parcourir tout le mot généré par la règle).
D'où une complexité en $\mathcal{O}(|\mathcal{V}| + |\mathcal{R}| \times m)$.

29. (a) Si $\mathcal{G}$ est élémentaire, il n'existe qu'une seule règle de la forme $S \to \alpha$ pour démarrer un arbre de dérivation. Puis, pour chaque X qui apparait dans α, il n'y a qu'une seule règle de la forme $X \to \alpha'$ pour continuer à générer un sous-arbre de dérivation. Ainsi, il n'y a jamais plusieurs choix possibles quand on essaye de continuer l'arbre de dérivation, et le processus s'arrête quand plus aucune variable n'est présente dans plus aucune feuille. Le mot v ainsi obtenu (si le processus s'arrête !) est alors unique, et l'arbre de dérivation obtenu est le seul possible. Donc la grammaire n'est pas ambigüe (et on a même que $|\mathcal{L}(\mathcal{G})| \leq 1$).

(b) Une grammaire acyclique peut être ambigüe. Par exemple :

$$\begin{aligned} S &\to X \mid Y \\ X &\to a \\ Y &\to Z \\ Z &\to a \end{aligned}$$

30. On reprend le raisonnement de la question 29a.

- Si $\mathcal{G}$ est acyclique, alors le processus s'arrête, et on fini par générer un unique mot $v \in \Sigma^\star$. Donc $|\mathcal{L}(\mathcal{G})| = 1$.
- Sinon, comme $\mathcal{G}$ est accessible, on va forcément finir par générer une variable X concernée par le cycle, et on n'arrivera jamais à terminer l'arbre de dérivation. Dans ce cas, on a $\mathcal{L}(\mathcal{G}) = \varnothing$.

31.
```
let ordre_topologique g =
  let n = Array.length g in
  let deja_vu = Array.make n false in
  let topo = ref [] in
  let rec parcours i = match deja_vu.(i) with
    | true -> ()
    | false ->
      deja_vu.(i) <- true;
      let traiter_symb s = match s with
        | T _ -> ()
        | V j -> parcours j
      in
      List.iter traiter_symb g.(i) ;
      topo := i :: !topo
  in
  for i = 0 to n-1 do
    parcours i
  done ;
  !topo
```

32. (a) Pour chaque $X \in \mathcal{V}$, on calcule le seul mot généré depuis X, et on mémoïse les résultats dans un tableau pour obtenir un algorithme efficace.

```
let engendre g =
  let n = Array.length g in
  let gen = Array.make n None in
  let rec genere i = match gen.(i) with
    | Some s -> s
    | None ->
      let traiter_symb = function
        | T c -> String.make 1 c
        | V j -> genere j
      in
      let concatener mot symb = mot ^ traiter_symb symb in
      let s = List.fold_left concatener "" g.(i) in
      gen.(i) <- Some s ;
      s
  in
  genere 0
```

(b) chaque appel récursif va effectuer au plus m concaténations, avec des mots de taille au plus $|v|$. Et grâce à la mémoïsation, on effectue un appel récursif par variable.
D'où une complexité totale en $\mathcal{O}(m \times |v| \times |\mathcal{V}|)$.

2.2.2 Complexité grammaticale de Kolmogorov

33. Soit $\mathcal{G}$ telle que $\mathcal{L}(\mathcal{G}) = \{v\}$ et $|\mathcal{G}| = K(v)$.

- Si $\mathcal{G}$ n'est pas accessible, on pourrait supprimer les variables non accessibles et les règles de production les concernant sans changer $\mathcal{L}(\mathcal{G})$, et on obtiendrait une grammaire plus petite (pas forcément strictement si les règles ainsi supprimées sont toutes de la forme $X \to \varepsilon$).
Donc on peut supposer sans perte de généralité que $\mathcal{G}$ est accessible.

- Si $\mathcal{G}$ n'est pas élémentaire, elle possède deux règles de la forme $X \to \alpha \mid \beta$, et :
  - on peut forcément dériver un mot de Σ^+ depuis α ou β, car sinon la règle en question serait inutile pour générer v, et on pourrait construire une grammaire plus petite ;
  - de plus, on a $\alpha \Rightarrow^* v'$ et $\beta \Rightarrow^* v'$ avec le même sous-mot $v' \neq \varepsilon$ de v (car sinon, notre grammaire pourrait dériver deux mots distincts).

  Donc dans ce cas, on pourrait supprimer l'une des deux règles sans changer $\mathcal{L}(\mathcal{G})$, et on obtiendrait une grammaire plus petite.
  Donc on peut supposer sans perte de généralité que $\mathcal{G}$ est élémentaire.
- Ainsi, d'après la question 30, on peut également supposer que $\mathcal{G}$ est acyclique.

34. Soit $\mathcal{G}_u = (\Sigma, \mathcal{V}_u, \mathcal{R}_u, S_u)$ et $\mathcal{G}_v = (\Sigma, \mathcal{V}_v, \mathcal{R}_v, S_v)$ telles que $\mathcal{L}(\mathcal{G}_u) = \{u\}$ et $|\mathcal{G}_u| = K(u)$ et $\mathcal{L}(\mathcal{G}_v) = \{v\}$ et $|\mathcal{G}_v| = K(v)$ (avec $\mathcal{V}_u \cap \mathcal{V}_v = \varnothing$). On pose $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{R}, S)$ avec S un nouveau symbole et :
    - $\mathcal{V} = \mathcal{V}_u \cup \mathcal{V}_v \cup \{S\}$;
    - $\mathcal{R} = \mathcal{R}_u \cup \mathcal{R}_v \cup \{S \to S_u S_v\}$.

    On obtient ainsi une grammaire $\mathcal{G}$ telle que $\mathcal{L}(\mathcal{G}) = \{uv\}$ et $|\mathcal{G}| = K(u) + K(v) + 2$.
    D'où $K(uv) \leq K(u) + K(v) + 2$.
35. On utilise le principe de l'exponentiation rapide.
    Soit $v \in \Sigma^\star$ et $k \in \mathbb{N}^\star$, et soit $\mathcal{G} = (\Sigma, \mathcal{V}, \mathcal{R}, S)$ telle que $\mathcal{L}(\mathcal{G}) = \{v\}$ et $|\mathcal{G}| = K(v)$.
    On construit par récurrence une grammaire $\mathcal{G}_k$ telle que $\mathcal{L}(\mathcal{G}_k) = \{v^k\}$:
    - $\mathcal{G}_1 = \mathcal{G}$;
    - pour $k > 1$, on pose $d = \lfloor \frac{k}{2} \rfloor$. Alors, en notant $\mathcal{G}_d = (\Sigma, \mathcal{V}_d, \mathcal{R}_d, S_d)$ et en posant S_k une nouvelle variable :
      - si $k = 2d$, on pose $\mathcal{G}_k = (\Sigma, \mathcal{V}_d \cup \{S_k\}, \mathcal{R}_d \cup \{S_k \to S_d S_d\}, S_k)$;
      - si $k = 2d + 1$, on pose $\mathcal{G}_k = (\Sigma, \mathcal{V}_d \cup \{S_k\}, \mathcal{R}_d \cup \{S_k \to S_d S_d S\}, S_k)$.

    On obtient bien ainsi une grammaire $\mathcal{G}_k$ telle que $\mathcal{L}(\mathcal{G}_k) = \{v^k\}$ et $|\mathcal{G}_k| \leq |\mathcal{G}_d| + 3$.
    Ainsi, par récurrence, on obtient que $|\mathcal{G}_k| \leq |\mathcal{G}| + 3\log_2(k)$.
    Donc $K(v^k) \leq K(v) + \mathcal{O}(\log k)$.
36. (a) Si $\alpha_i \notin \Sigma^\star$, considérons X_j qui maximise $|\mathcal{G}(X_j)|$ parmi les X_j qui apparaissent dans α_i. Chaque lettre de α_i peut être dérivée en un mot de taille au plus $\max(1, |\mathcal{G}(X_j)|)$ (le max est nécessaire si $\mathcal{G}(X_j) = \varepsilon$ et qu'il y a des lettres de Σ dans α_i).
    On obtient alors que $\mathcal{G}(X_i) \leq |\alpha_i| \times \max(|\mathcal{G}(X_j)|, 1)$.

    (b) Si on suppose que $(X_0, \ldots, X_{n-1})$ est un ordre topologique, alors dans la question précédente, on a nécessairement $j > i$. On obtient alors par récurrence que $|\mathcal{G}(S)| \leq |\mathcal{G}(X_0)| \leq \prod_{i=0}^{n-1} \max(1, |\alpha_i|)$.

    (c) En utilisant la formule donnée dans l'indication, on obtient que $|v| = |\mathcal{G}(S)| \leq 3^{\frac{M}{3}}$ avec $M = \sum_{i=0}^{n-1} |\alpha_i| = |\mathcal{G}|$. Cela donne $3\log_3 |v| \leq |\mathcal{G}|$, donc $\log |v| = \mathcal{O}(K(v))$.

2.2.3 Plus petite grammaire

37. On prend comme ensemble de certificats le type `grammaire` introduit avant la question 31 (on travaille donc uniquement avec des grammaires élémentaires), et f_{verif} doit :
    - vérifier que la grammaire $\mathcal{G}$ est bien de taille $\leq k$ (sinon, f_{verif} doit renvoyer `false` sans finir de calculer sa taille) ;
      ↪ dans ce cas, le certificat est bien de taille polynomial, et toutes les vérifications effectuées en temps polynomial en la taille de la grammaire se feront bien aussi en temps polynomial en $|v|$;
      ↪ cette vérification ne se fait pas en temps polynomial en la taille de k si k est écrit en binaire, mais elle se fait bien en temps polynomial en $|v|$ car $k \leq |v|$;

- vérifier que $\mathcal{G}$ est bien acyclique et accessible (ce qui se fait bien en temps polynomial en $|\mathcal{G}|$) ;
- si c'est bien le cas, alors d'après la question 32 on peut vérifier en temps polynomial que $\mathcal{L}(\mathcal{G}) = \{v\}$.

Ainsi, PPG $\in$ NP.

38. Supposons que $S' \subseteq S$ soit une couverture de G avec $|S'| = k$.
On construit une grammaire $\mathcal{G}$ avec les règles suivantes :
- $\forall i \in [\![1, p]\!],\ X_i \to \#s_i$;
- $\forall i \in [\![1, p]\!],\ Y_i \to s_i\#$;
- $\forall s_i \in S',\ Z_i \to X_i\#$.

Il reste alors la règle du symbole initial qu'on va noté X_0. On pose :

$$\forall i \in [\![1, p]\!],\ \alpha_i = \begin{cases} X_i a_i Y_i b_i X_i c_i Y_i d_i Z_i e_i & \text{si } s_i \in S' \\ X_i a_i Y_i b_i X_i c_i Y_i d_i X_i \# e_i & \text{sinon} \end{cases}$$

$$\forall j \in [\![1, q]\!],\ (a_j = s \text{—} t \in A),\ \beta_j = \begin{cases} Z_i Y_k a_j & \text{si } s_j \in S' \\ X_i Z_k a_j & \text{sinon} \end{cases}$$

On ajoute alors la règle : $X_0 \to \alpha_1 \cdots \alpha_p \beta_1 \cdots \beta_q$.
Comme S' est une couverture des sommets de G, on a bien que $X_0 \Rightarrow^* v$, et on a :

$$|\mathcal{G}| = 2p + 2p + 2k + 10k + 11(p - k) + 3q = 15p + 3q + k$$

Donc $K(v) \leq 15p + 3q + k$.

39. On remarque que les symboles qui ne sont ni un $s \in S$ ni un # n'apparaissent qu'une seule fois dans v. Ainsi, on ne fait pas diminuer $|\mathcal{G}|$ si on suppose que ces lettres n'apparaissent que dans la règle initiale $X_0 \to \alpha$. De plus, les facteurs qui apparaissent plusieurs fois sont tous de la forme $\#s$ ou $s\#$ ou $\#s\#$ pour un certain $s \in S$, on en déduit donc que les autres X_i avec $i \in [\![1, n-1]\!]$ ne peuvent dériver qu'un mot de cette forme.

40. On suppose que la grammaire est de la forme décrite dans la question précédente.
Supposons que pour un certain $s \in S$, il n'existe aucun $i \in [\![1, n-1]\!]$ tel que $\mathcal{G}(X_i) = \#s$. Comme $\#s$ est un facteur qui apparait forcément au moins 2 fois dans v sans # juste derrière, on en déduit que dans la règle initiale $X_0 \to \alpha$, $\#s$ apparait au moins deux fois dans α. Dans ce cas, on pourrait remplacer ces occurrences de $\#s$ par une nouvelle variable X et rajouter la règle $X \to \#s$, ce qui n'augmenterait pas $|\mathcal{G}|$ (on économise au moins 2 lettres dans α et on rajoute une règle de taille 2).
Le raisonnement est identique pour $s\#$.

41. On pose $S' = \{s \in S \mid \exists i \in [\![1, n-1]\!],\ \mathcal{G}(X_i) = \#s\#\}$.
S'il existe $a_j = s \text{—} t \in A$ telle que $s \notin S'$ et $t \notin S'$, alors pour générer le facteur $w_j = \#s\#t\#a_j$ il faut nécessairement générer le s et le t avec deux # en utilisant deux règles de la forme de la question précédente, et générer le # manquant et le a_j avec la règle initiale.
Dans ce cas, notons X_i tel que $\mathcal{G}(X_i) = \#s$ et X_j tel que $\mathcal{G}(X_j) = t\#$, et on peut :
- rajouter une règle $X \to X_i\#$ avec X une nouvelle variable ;
- remplacer les symboles qui génèrent w_j par $XX_k a_j$;
- remplacer par X la partie qui génère $\#s\#$ dans le mot $v_{i'}$ avec $s_{i'} = s$.

Cette modification de change pas $|\mathcal{G}|$, et on a alors rajouté un X tel que $\mathcal{G}(X) = \#s\#$, ce qui rajoute s dans l'ensemble S' considéré.
Finalement, on obtient que S' est une couverture de G, et la grammaire $\mathcal{G}$ qui génère v est de taille :

$$|\mathcal{G}| = 15p + 3q + |S'|$$

Donc si $K(v) \leq 15p + 3q + k$ pour un certain k, alors G possède une couverture de taille k.

42. Le mot v se construit bien en temps polynomial en $|G|$, ainsi que l'entier $15p + 3q + k$.
Et d'après les questions précédentes, (G, k) est une instance positive de VertexCover si et seulement si $(v, 15p + 3q + k)$ est une instance positive de PPG. Donc VertexCover $\leq_{\mathsf{P}}$ PPG.
Or, d'après la question 25, VertexCover est NP-difficile, donc PPG est NP-difficile.
De plus, d'après la question 37, PPG $\in$ NP. Donc PPG est NP-complet.