I Lemme de l'étoile pour les langages hors-contexte

On admet la version suivante du lemme de l'étoile pour les langages hors-contextes :

Théorème : Lemme de l'étoile hors-contexte

Si L est un langage hors-contexte alors il existe un entier n tel que, pour tout mot $t \in L$ tel que $|t| \geq n$, il existe $u, v, w, x, y \in \Sigma^*$ tels que $t = uvwxy$ avec :

- $|vwx| \leq n$;
- $vx \neq \varepsilon$;
- $\forall i \in \mathbb{N}, uv^iwx^iy \in L$.

Soient $L_1 = \{a^nb^nc^p \mid n, p \in \mathbb{N}\}$ et $L_2 = \{a^nb^nc^n \mid n \in \mathbb{N}\}$.

1. Montrer que L_1 est un langage hors-contexte.

> Solution : L_1 est engendré par la grammaire avec les règles $S \longrightarrow XY$, $X \longrightarrow aXb \mid \varepsilon$ et $Y \longrightarrow cY \mid \varepsilon$.

2. Montrer que L_2 n'est pas un langage hors-contexte.

> Solution : Supposons par l'absurde que L_2 est un langage hors-contexte et soit n l'entier donné par le lemme de l'étoile hors-contexte.
> Soit $t = a^nb^nc^n$. Comme $t \in L_2$ et $|t| \geq n$, on peut écrire $t = uvwxy$ avec $|vwx| \leq n$, $vx \neq \varepsilon$ et $\forall i \in \mathbb{N}, uv^iwx^iy \in L_2$.
> Comme $|vwx| \leq n$, vwx ne peut pas contenir à la fois des a et des c. Supposons par exemple que vwx ne contienne pas de c. Alors uv^2wx^2y contient n c mais un nombre de a ou b strictement supérieur à n (car $vx \neq \varepsilon$), ce qui contredit L_2.

3. Montrer que l'ensemble des langages hors-contextes n'est pas stable par intersection ni par passage au complémentaire.

> Solution : Soit $L_3 = \{a^pb^nc^n \mid n, p \in \mathbb{N}\}$. L_3 est hors-contexte (grammaire similaire à L_1) mais $L_1 \cap L_3 = L_2$ n'est pas hors-contexte.
> Comme les langages hors-contextes sont stables par union (voir cours), s'ils étaient stables par complémentaire alors $L_1 \cap L_3 = \overline{\overline{L_1} \cup \overline{L_3}}$ serait hors-contexte, ce qui est faux.

II Algorithmes de Borůvka

Soit $G = (S, A)$ un graphe non-orienté connexe et pondéré par $w : A \longrightarrow \mathbb{R}$. On note $n = |S|$ et $p = |A|$. On suppose que tous les poids de G sont distincts (c'est-à-dire : w injective) et que $S = \{0, ..., n-1\}$.

II.1 Théorie

1. Montrer que G possède un arbre couvrant de poids minimum.

> Solution : Étant connexe, G possède bien un arbre couvrant, obtenu par exemple avec un arbre de parcours en profondeur (constitué de toutes les arêtes parcourues dans le DFS).
> Ainsi l'ensemble $E = \{w(T) \mid T \text{ arbre couvrant de } G\}$ est non vide et fini donc il possède un minimum.

2. Montrer que G possède un unique arbre couvrant de poids minimum.

> Solution : Supposons par l'absurde que G possède deux arbres couvrants T et T' de poids minimum. Soit e l'arête de poids minimum appartenant à exactement un de ces deux arbres. Supposons par exemple que e appartient à T. Comme il contient n sommets et n arêtes, $T' + e$ contient un cycle C. T n'a pas de cycle donc C contient une arête e' qui n'appartient pas à T.
> Soit $T'' = T' + e - e'$. Alors :
>
> - T'' est connexe. En effet, si $u, v \in S$ alors il existe un chemin P de u à v dans T'. Si P passe par e', on remplace e' par le reste du chemin dans C. Ainsi, il existe un chemin de u à v dans T''.
> - T'' est un arbre couvrant car il contient $n-1$ arêtes et est connexe.
> - $w(T'') = w(T') + w(e) - w(e') < w(T')$ car $w(e) < w(e')$, tous les poids étant distincts.

T'' contredit l'hypothèse de minimalité de T'. Ainsi, G possède bien un unique arbre couvrant de poids minimum.

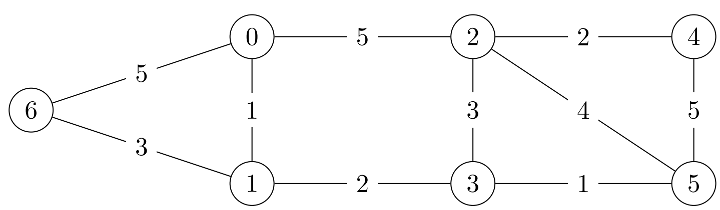
On appelle T^* l'unique arbre couvrant de poids minimum de G.
Soit $X \subset S$. On dit qu'une arête est sûre pour X si elle est de poids minimum parmi les arêtes ayant exactement une extrémité dans X. Autrement dit, une arête e est sûre pour X si $w(e) = \min\{w(e') \mid \{u, v\} \in A, u \in X, v \notin X\}$.
L'objectif de l'algorithme de Borůvka est de construire un arbre couvrant de poids minimum T en conservant une partition F de S, correspondant aux composantes connexes de T.
À chaque étape, on ajoute une arête sûre pour chaque composante connexe de F :

Algorithme de Borůvka

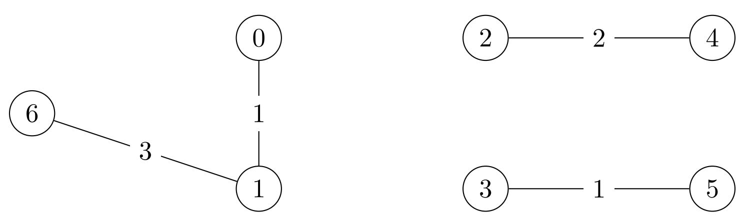
$F \longleftarrow \{\{x\} \mid x \in S\}$
$T \longleftarrow \emptyset$
Tant que $|F| > 1$ **:**
    $E \longleftarrow \emptyset$
    Pour $C \in F$ **:**
        $e \longleftarrow$ arête sûre pour C
        $E \longleftarrow E \cup \{e\}$
    $F \longleftarrow$ partition de S obtenue en fusionnant les composantes connexes de F avec les arêtes de E
    $T \longleftarrow T \cup E$
Renvoyer T

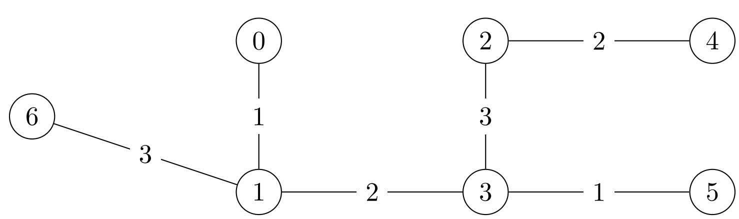
L'étape de fusion des composantes connexes consiste, pour chaque arête $e = \{u, v\}$ de E, à remplacer dans F les composantes connexes C_1 et C_2 contenant u et v par leur union $C_1 \cup C_2$.

3. Appliquer l'algorithme de Borůvka sur le graphe suivant, en donnant à chaque l'ensemble des arêtes de T à la fin de chaque passage dans la boucle **Tant que** :



Solution :





4. Montrer que l'algorithme de Borůvka termine, en utilisant un variant de boucle.

Solution : $|F|$ est strictement décroissant à chaque itération de la boucle **Tant que** et $|F| \geq 0$ donc l'algorithme termine.

5. Soit $X \subset S$ et e une arête sûre pour X. Montrer que T^* contient e.

Solution : Notons $e = \{u, v\}$ et supposons que T^* ne contienne pas e.
$T^* + e$ contient un cycle C car il a n arêtes pour n sommets.
Soit $e^* \neq e$ une arête de C ayant exactement une extrémité dans X. $w(e) < w(e^*)$ car e est sûre pour X.
$T^* + e - e^*$ contient $n - 1$ arêtes et est acyclique (s'il possédait un cycle C', on pourrait y remplacer e par le reste du cycle dans C pour obtenir un cycle dans T^*).
Ainsi, $T^* + e - e^*$ est un arbre couvrant de poids strictement inférieur à T^*, ce qui est absurde.

6. Montrer que l'algorithme de Borůvka renvoie bien T^*.

Solution : L'algorithme ne rajoute dans T que des arêtes appartenant à T^*, donc T est toujours un sous-ensemble de T^*. De plus, T ne contient qu'une composante connexe à la fin donc $T = T^*$.

II.2 Implémentation

On va utiliser une structure d'Union-Find pour représenter les composantes connexes de F, sous la forme d'un tableau `uf` de taille n tel que `uf.(x)` soit le père de `x` dans l'arbre contenant `x`. Si `x` est une racine, `uf.(x)` contiendra `x`.
On n'utilisera pas d'optimisation de type union par rang ou compression de chemin.

7. Écrire une fonction `create : int -> int array` telle que `create n` renvoie un tableau de taille `n` initialisé avec les entiers de 0 à `n - 1`.

Solution :

```
let create n =
  let uf = Array.make n 0 in
  for i = 0 to n - 1 do
    uf.(i) <- i
  done;
  uf
```

8. Écrire une fonction `find : int array -> int -> int` telle que `find uf x` renvoie la racine de l'arbre contenant `x` dans la structure d'Union-Find représentée par le tableau `uf`.

Solution :

```
let rec find uf i =
    if uf.(i) = i then i
    else find uf uf.(i)
```

9. Écrire une fonction `union : int array -> int -> int -> unit` telle que `union uf x y` fusionne les composantes connexes de `x` et `y` dans la structure d'Union-Find représentée par le tableau `uf`.

Solution :

```
let union uf x y =
    let rx = find uf x in
    let ry = find uf y in
    uf.(rx) <- ry
```

10. Écrire une fonction `meme_cc : int array -> int -> int -> bool` telle que `meme_cc uf x y` détermine si `x` et `y` sont dans la même composante connexe.

Solution :

```
let meme_cc uf i j =
    find uf i = find uf j
```

11. Écrire une fonction `n_cc : int array -> int` telle que `n_cc uf` renvoie le nombre de composantes connexes dans `uf`.

Solution :

```
let n_cc uf =
  let n = Array.length uf in
  let ans = ref 0 in
  for i = 0 to n - 1 do
    if uf.(i) = i then incr ans
  done;
  !ans
```

Le graphe G est représenté par une liste d'adjacence `g` telle que `g.(i)` contient une liste des arêtes partant de `i`, où chaque arête est un couple (w, j) où w est le poids de l'arête et i et j les extrémités de l'arête.

12. Écrire une fonction `aretes_sures : (float * int) list array -> int array -> (float * int * int) array` telle que
`aretes_sures g uf` renvoie un tableau `ans` de taille n où, si `i` est une racine dans `uf`, `ans.(i)` contient l'arête sûre pour la composante connexe de `i`.
Si `i` n'est pas une racine, `ans.(i)` contiendra `(max_float, -1, -1)`.

Solution : On parcourt chaque arête et on met à jour l'arête sûre de la composantes connexe si elle est plus petite.

```
let aretes_sures g uf =
  let n = Array.length g in
  let ans = Array.make n (max_float, -1, -1) in
  for u = 0 to n - 1 do
    let rec aux = function
      | [] -> ()
      | (w, v) :: q -> if not (meme_cc uf u v) then (
        let e = (w, u , v) in
        let e_ = ans.(find uf u) in
        if e_ > e then ans.(find uf u) <- e;
        let e_ = ans.(find uf v) in
        if e_ > e then ans.(find uf v) <- e
      );
      aux q in
    aux g.(u)
  done;
  ans
```

13. Écrire une fonction `boruvka : (float * int) list array -> (float * int) list array` renvoyant l'arbre couvrant de poids minimum de G par l'algorithme de Borůvka.

Solution :

```
let boruvka g =
  let n = Array.length g in
  let uf = create n in
  let t = Array.make n [] in
  while n_cc uf > 1 do
    let ans = aretes_sures g uf in
    for i = 0 to n - 1 do
      let (w, u, v) = ans.(i) in
      if u <> -1 then (
        t.(u) <- (w, v) :: t.(u);
        t.(v) <- (w, u) :: t.(v);
        union uf u v
      )
    done
  done;
  t;;
```

14. Quitte à utiliser l'optimisation par compression de chemin et union par rang, on suppose que `union` et `find` sont en O(1). Montrer que la complexité de l'algorithme de Borůvka est en $O(p \log n)$. Comparer avec l'algorithme de Kruskal.

Solution : À chaque étape, $|F|$ est divisé au moins par deux donc l'algorithme s'arrête en $O(\log n)$ étapes.
À chaque étape, on parcourt en O(p) toutes les arêtes.
La complexité est donc en $O(p \log n)$, comme l'algorithme de Kruskal.

III Théorème de Chomsky-Schützenberger

III.1 Langage de Dyck

Soit $n \in \mathbb{N}^*$. On définit $\Sigma_n = \{a_1, \overline{a_1}, a_2, \overline{a_2}, ..., a_n, \overline{a_n}\}$. Les lettres a_i seront appelées parenthèses ouvrantes et les $\overline{a_i}$ sont les parenthèses fermantes.
Soit $G_n = (\Sigma_n, \{S\}, R, S)$, où R contient les règles :

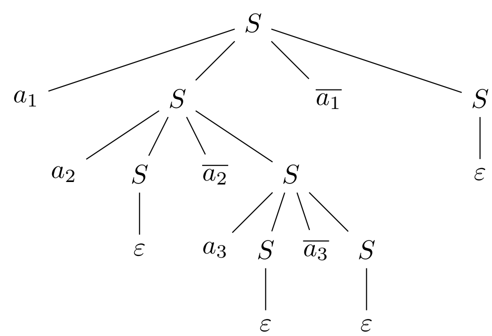
$$S \longrightarrow a_1 S \overline{a_1} S \mid a_2 S \overline{a_2} S \mid ... \mid a_n S \overline{a_n} S \mid \varepsilon$$

On définit le langage D_n de Dyck d'ordre n comme celui engendré par G_n.

On note Pref(u) l'ensemble des préfixes d'un mot u.

1. Représenter graphiquement un arbre de dérivation de $u = a_1 a_2 \overline{a_2} a_3 \overline{a_3} \overline{a_1}$ pour G_3.

Solution :



2. Soit $L = \{u \in \Sigma_1^* \mid \forall v \in \text{Pref}(u), |v|_{a_1} \geq |v|_{\overline{a_1}} \text{ et } |u|_{a_1} = |u|_{\overline{a_1}}\}$. Montrer que $D_1 = L$.

Solution :

$D_1 \subset L$: Soit H_n : « Si $S \Rightarrow^n u \in \Sigma_1^*$ alors $u \in L$ ».
H_1 est vraie car $\varepsilon \in L$.
Soit $n \geq 2$. Supposons H_k vraie pour $k < n$. Soit $u \in \Sigma_1^*$ tel que $S \Rightarrow^n u$.
On a alors $S \to a_1 S \overline{a_1} S \Rightarrow^{n-1} a_1 v \overline{a_1} w = u$ avec $S \Rightarrow^k v$ et $S \Rightarrow^{n-k} w$.
D'après l'hypothèse de récurrence, $v \in L$ et $w \in L$.
Comme $|v|_{a_1} = |v|_{\overline{a_1}}$ et $|w|_{a_1} = |w|_{\overline{a_1}}$, on a $|u|_{a_1} = |v|_{a_1} + 1 + |w|_{a_1} = |v|_{\overline{a_1}} + 1 + |w|_{\overline{a_1}} = |u|_{\overline{a_1}}$.
On peut montrer aussi que tout préfixe de u vérifie la propriété. Ainsi, $u \in L$.

$L \subset D_1$: Soit H_n : « Si $u \in L$ et $|u| = n$ alors $u \in D_1$ ».
H_0 est vraie car $\varepsilon \in D_1$.
Soit $n \geq 1$. Supposons H_k vraie pour $k < n$. Soit $u = u_1 ... u_n \in L$.
Notons $i = \min\{j \mid |u_1...u_j|_{a_1} = |u_1...u_j|_{\overline{a_1}}\}$ (i est bien défini car $|u_1...u_n|_{a_1} = |u_1...u_n|_{\overline{a_1}}$).
Soit $v = u_1...u_i$ et $w = u_{i+1}...u_n$. On peut montrer que $v \in L$ et $w \in L$.
D'après l'hypothèse de récurrence, $S \Rightarrow^* v$ et $S \Rightarrow^* w$.
Donc $S \to a_1 S \overline{a_1} S \Rightarrow^* a_1 v \overline{a_1} S \Rightarrow^* a_1 v \overline{a_1} w = u$.

3. En déduire que D_1 n'est pas régulier.

Solution : Supposons D_1 régulier et soit n l'entier donné par le lemme de l'étoile.
Soit $u = a_1^n \overline{a_1}^n$. Comme $u \in D_1$ et $|u| \geq n$, on peut écrire $u = xyz$ avec $|y| \geq 1$, $|xy| \leq n$ et $\forall i \in \mathbb{N}, xy^i z \in D_1$.
Comme $|xy| \leq n$, y ne peut contenir que des a_1. Donc xy^2z contient plus de a_1 que de $\overline{a_1}$: contradiction.

4. Montrer que $D_2 \neq \{u \in \Sigma_n \mid \forall v \in \text{Pref}(u), \forall i \in [\![1, 2]\!], |v|_{a_i} \geq |v|_{\overline{a_i}} \text{ et } |u|_{a_i} = |u|_{\overline{a_i}}\}$?

> Solution : Soit $u = a_1 a_2 \overline{a_1}\,\overline{a_2}$.
> $|u|_{a_1} = |u|_{\overline{a_1}} = 1$ et $\forall v \in \text{Pref}(u), |v|_{a_1} \geq |v|_{\overline{a_1}}$
> Mais $u \notin D_2$ car si $S \Rightarrow^* u$ alors $S \Rightarrow a_1 S \overline{a_1} S \Rightarrow^* a_1 a_2 \overline{a_1}\,\overline{a_2}$ ce qui est impossible.

On définit le type suivant :

```
type lettre = O of int | F of int
```

tel qu'une lettre a_i sera représentée par `O i` et une lettre $\overline{a_i}$ par `F i`.

5. Écrire une fonction `dyck : lettre list -> bool` qui détermine si un mot u est un mot d'un langage de Dyck D_n pour un certain $n \in \mathbb{N}$, en complexité linéaire en la taille de u.
   On pourra utiliser une pile (sous forme d'une liste OCaml).

> Solution : On utilise une pile (sous forme de liste) pour vérifier si le mot est bien parenthésé :
>
> - Si on lit a_i, on l'ajoute à la pile.
> - Si on lit $\overline{a_i}$, on vérifie que le sommet de la pile est a_i et on le retire.
>
> Si la pile est vide à la fin, le mot est bien parenthésé.
>
> ```
> let dyck u =
>     let rec aux u pile = match u, pile with
>         | [], [] -> true
>         | [], _ -> false
>         | O i :: q, _ -> aux q (i :: pile)
>         | F i :: q, [] -> false
>         | F i :: q, j :: p -> i = j && aux q p in
>     aux u []
> ```

Soient Σ et Γ deux alphabets. On appelle morphisme de mots une fonction $\varphi : \Sigma^* \longrightarrow \Gamma^*$ telle que pour tout $u, v \in \Sigma^*$, $\varphi(uv) = \varphi(u)\varphi(v)$.
Remarque : il suffit de définir un morphisme de mots sur les lettres, car si $u = u_1 ... u_n \in \Sigma^*$ alors $\varphi(u) = \varphi(u_1)...\varphi(u_n)$.
Si L est un langage, on note $\varphi(L) = \{\varphi(u) \mid u \in L\}$.

6. Montrer que si φ est un morphisme de mots, alors $\varphi(\varepsilon) = \varepsilon$.

> Solution : Soit $u = \varphi(\varepsilon)$. Alors $u = \varphi(\varepsilon) = \varphi(\varepsilon\varepsilon) = \varphi(\varepsilon)\varphi(\varepsilon) = u^2$. D'où $|u| = |u|^2 = 2|u|$. Donc $|u| = 0$ et $u = \varepsilon$.

7. Donner sans justification une expression régulière de $\varphi(D_1)$ pour le morphisme de mots φ défini par $\varphi(a_1) = aa$ et $\varphi(\overline{a_1}) = \varepsilon$.

> Solution : Il s'agit de $(aa)^*$.

8. Soit φ un morphisme de mots. Montrer que si L est un langage hors-contexte alors $\varphi(L)$ est un langage hors-contexte.

> Solution : Soit $G = (\Sigma, V, R, S)$ une grammaire hors-contexte engendrant L.
> On étend φ aux variables en posant $\varphi(X) = X$ pour $X \in V$.
> On définit alors $G' = (\Gamma, V, R', S)$ où $R' = \{X \longrightarrow \varphi(u) \mid X \longrightarrow u \in R\}$.
> On montre alors que $L(G') = \varphi(L(G))$.

III.2 Théorème de Chomsky-Schützenberger

Soit $G = (\Sigma, V, R, S)$ une grammaire hors-contexte. On dit que G est en forme normale de Chomsky si toutes les règles de production sont de l'une des formes suivantes :

- $X \longrightarrow a$, avec $a \in \Sigma$
- $X \longrightarrow YZ$, avec $Y, Z \in V$

On admet que si G est une grammaire quelconque, alors il existe une grammaire G' en forme normale de Chomsky telle que $L(G') = L(G) \setminus \{\varepsilon\}$.

9. Donner sans justification le langage engendré par la grammaire G_0 en forme normale de Chomsky définie par les règles suivantes :

   - $S \longrightarrow AX \mid AB$
   - $X \longrightarrow SB$
   - $A \longrightarrow a$
   - $B \longrightarrow b$

   > Solution : On peut montrer par double inclusion que $L(G_0) = \{a^n b^n \mid n \in \mathbb{N}\}$.

On veut démontrer :

Théorème : Chomsky-Schützenberger

Un langage L est hors-contexte si et seulement il existe un langage régulier K, un langage de Dyck D_n et un morphisme de mots φ tels que $L = \varphi(D_n \cap K)$.

10. On suppose L hors-contexte et K régulier sur un même alphabet Σ. En considérant $A = (\Sigma, Q, \delta, q_0, F)$ un automate fini déterministe reconnaissant K et $G = (\Sigma, V, R, S)$ une grammaire en forme normale de Chomsky engendrant L, montrer que $L \cap K$ est un langage hors-contexte.
    Pour cela, on construira une grammaire G' ayant un symbole initial S' et une variable $X_{p,q}$ pour toute variable $X \in V$ et tout états $p, q \in Q$.

    > Solution : On pose $G' = (\Sigma, V', R', S')$ où $V' = \{S'\} \cup \{X_{p,q} \mid X \in V, p, q \in Q\}$ et R' contient les règles :
    >
    > - $S' \longrightarrow S_{q_0,q_f}$, pour tout état $q_f \in F$;
    > - $X_{q,\delta(q,a)} \longrightarrow a$, pour toute règle $X \longrightarrow a \in R$ et tout $q \in Q$;
    > - $X_{p,q} \longrightarrow Y_{p,r} Z_{r,q}$, pour tout règle $X \longrightarrow YZ \in R$ et tout $p, q, r \in Q$.

11. En déduire un sens du théorème.

    > Solution : Supposons $L = \varphi(D_n \cap K)$ avec K régulier. D'après la question précédente, $D_n \cap K$ est hors-contexte. Donc L est hors-contexte d'après la question 8.

Soit $G = (\Sigma, V, R, S)$ une grammaire hors-contexte en forme normale de Chomsky. On numérote les règles de la forme $X \longrightarrow YZ$ par $r_1, r_2, ..., r_k$. On pose $G' = (\Sigma', V, R', S)$ où :

- $\Sigma' = \Sigma \cup \{\overline{a} \mid a \in \Sigma\} \cup \displaystyle\bigcup_{i=1}^{k} \{a_i, \overline{a_i}, b_i, \overline{b_i}, c_i, \overline{c_i}\}$;
- $R' = \{X \longrightarrow a_i b_i Y \overline{b_i} c_i Z \overline{c_i} \overline{a_i}, \text{pour } i \in [\![1, k]\!] \text{ et } r_i = X \longrightarrow YZ\} \cup \{X \longrightarrow a\overline{a}, \text{pour } X \longrightarrow a \in P\}$.

12. Montrer qu'il existe n tel que $L(G') \subset D_n$.

    > Solution : Avec les notations de l'énoncé, on pose $n = 3k + |\Sigma|$ et $\Sigma_n = \Sigma'$. Montrons un résultat plus fort, c'est-à-dire que pour $X \in V$, si $X \Rightarrow^\ell u$ avec $u \in \Sigma'^*$, alors $u \in D_n$, par récurrence sur la taille des dérivations ℓ :
    >
    > - si $\ell = 1$, alors la dérivation est de la forme $X \longrightarrow a\overline{a}$, avec $a \in \Sigma$. Dès lors, $S_n \Rightarrow aS_n\overline{a}S_n \Rightarrow a\overline{a}S_n \Rightarrow a\overline{a}$ est une dérivation de u dans G_n (on a renommé le symbole de départ en S_n pour ne pas confondre avec S) ;
    > - si on suppose le résultat vrai pour toute dérivation de taille $< \ell$, avec $\ell > 1$ fixé, alors la première dérivation immédiate est de la forme : $X \Rightarrow a_i b_i Y \overline{b_i} c_i Z \overline{c_i} \overline{a_i}$. De plus, il existe v et w tels que $u = a_i b_i v \overline{b_i} c_i w \overline{c_i} \overline{a_i}$ et $Y \Rightarrow^* v$, $Z \Rightarrow^* w$. Par hypothèse de récurrence, v et w sont dans D_n. Dès lors, $S_n \Rightarrow a_i S_n \overline{a_i} S_n \Rightarrow^* a_i b_i S_n \overline{b_i} S_n \overline{a_i} c_i S_n \overline{c_i} S_n \overline{a_i} S_n \Rightarrow^* a_i b_i v \overline{b_i} \overline{a_i} c_i w \overline{c_i} \overline{a_i} = u$.
    >
    > On conclut par récurrence.

On pose $\varphi : \Sigma'^* \longrightarrow \Sigma^*$ le morphisme de mots défini par :

- pour $a \in \Sigma$, $\varphi(a) = a$ et $\varphi(\overline{a}) = \varepsilon$;
- pour $i \in [\![1, k]\!]$, $\varphi(a_i) = \varphi(\overline{a_i}) = \varphi(b_i) = \varphi(\overline{b_i}) = \varphi(c_i) = \varphi(\overline{c_i}) = \varepsilon$.

13. Montrer que $L(G) = \varphi(L(G'))$.

> Solution : On remarque que pour toute règle $X \longrightarrow \alpha \in R'$, alors $X \longrightarrow \varphi(\alpha) \in R$. Un raisonnement par récurrence et la définition des morphismes de mots permet donc de conclure que $\varphi(L(G')) \subset L(G)$. Réciproquement, pour toute règle $X \longrightarrow \alpha \in R$, il existe β tel que $X \longrightarrow \beta \in R'$ et $\alpha = \varphi(\beta)$. De la même manière, cela donne l'inclusion réciproque.

Pour L un langage sur un alphabet Σ, on note $P(L) \subset \Sigma$ l'ensemble des premières lettres des mots de L, $F(L) \subset \Sigma^2$ l'ensemble des facteurs de taille 2 des mots de L et $N(L) = \Sigma^2 \setminus F(L)$.

14. On pose $K = P(L(G'))\Sigma'^* \setminus \Sigma'^* N(L(G'))\Sigma'^*$. Montrer que K est un langage régulier.

> Solution : $P(L(G'))$ et $N(L(G'))$ sont réguliers car finis. Par concaténation et différence de langages réguliers, K est régulier.

15. Montrer le théorème de Chomsky-Schützenberger.

> Solution : Montrons que $L(G') = D_n \cap K$, où D_n est le langage défini à la question 12 et K le langage défini à la question précédente. On a déjà montré $L(G') \subset D_n$. De plus, $L(G') \subset R$, car tout mot de $L(G')$ commence par une première lettre d'un mot de $L(G')$ et ne contient aucun facteur de taille 2 de $N(L(G'))$. Cela montre la première inclusion.
>
> Pour $X \in V$, si on note $L(G', X)$ le langage engendré par (Σ', V, P', X) et $K_X = P(L(G', X))\Sigma'^* \setminus \Sigma'^* N(L(G', X))\Sigma'^*$, alors on peut montrer par récurrence sur la taille des mots que $D_n \cap R_X \subset L(G', X)$. Dès lors, $D_n \cap K = D_n \cap K_S \subset L(G', S) = L(G')$.
>
> Finalement, par la question 13, $L(G) = \varphi(L(G')) = \varphi(D_n \cap R)$.
>
> Pour conclure complètement, on remarque que si L est un langage hors-contexte, alors il existe G une grammaire en forme normale de Chomsky telle que $L \setminus \{\varepsilon\} = L(G)$. Dès lors quitte à considérer $K \cup \{\varepsilon\}$ si $\varepsilon \in L$, on obtient le résultat voulu.