Mines 2018 : option informatique
Recherche de motifs

1 Recherche naïve d'un motif

1. Considérons le motif $s = aaa$ et le texte $t = aaaaaa$. Les occurrences de s dans t sont $3, 4, 5, 6$ Ainsi $y = 3$ et $y' = 4$ sont des occurrences de s dans t, s est de taille $k = 2$ mais $y' - k + 1 = 3 \leq y$.
2. Si s est de taille k et t de taille n, par définition la première occurrence envisageable est k. Les occurrences possibles sont les éléments de $\{k, \dots, n\}$ et il y en a au plus $n - k + 1$. Cette borne est atteinte dans le cas d'un alphabet à un seul élément.
3. La longueur d'une liste non vide est égal à 1 plus la longueur de sa queue.

```
let rec longueur l = match l with
   |[] -> 0
   |x::q -> 1 + (longueur q);;
```

   En notant C_n le nombre d'opération pour une liste de taille n, on a $C_0 = O(1)$ et $C_{n+1} = O(1) + C_n$. La complexité est ainsi $O(n)$;, linéaire en fonction de la taille de la liste.
   C'est en fait la fonction `List.length` et nous utiliserons cette syntaxe dans la suite.
4. On peut, par exemple, filtrer sur le couple (s, t) et tester l'égalité des premières lettres puis travailler avec les queues.

```
let rec prefixe s t = match (s,t) with
   |[],_ -> true
   |_,[] -> false
   |a::qs,b::qt -> (a=b) && (prefixe qs qt);;
```

   La complexité est immédiatement $O(\min(n, k))$.
5. On cherche successivement si s est préfixe de t puis de t privé de sa première lettre puis de ses deux premières etc.
   On écrit pour cela une fonction auxiliaire locale `chercher_depuis : int → int list → int list`. Dans l'appel `chercher_depuis i t`, on donne les occurrences de s dans t en numérotant les éléments de t depuis i (ce qui moralement signifie que l'on a déjà enlevé les $i - 1$ premières lettres du texte de départ).
   Notons que si on trouve s préfixe de t depuis la position i, on trouve l'occurrence $i + k - 1$.

```
let recherche_naive s t =
   let k=List.length s in

   let rec chercher_depuis i t =
      match t with
     |[] -> []
     |x::q -> if prefixe s t then (i+k-1)::(chercher_depuis (i+1) q)
              else chercher_depuis (i+1) q

   in chercher_depuis 1 t ;;
```

6. La recherche naïve effectue au plus n tests de préfixe. Comme après chaque test on a un nombre constant d'opérations, la complexité est finalement $O(n \min(n, k))$.

2 Automates finis déterministes à repli

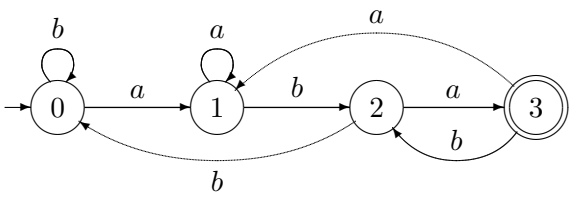
7. Par définition de ρ, pour tout $q \in Q_A$, la suite $(\rho^j(q))_{j\in\mathbb{N}}$ est d'une part composée d'entiers naturels et d'autre part décroissante.
   Etant minorée et décroissante, elle converge.
   Comme elle est à valeurs entières, elle stationne donc à partir d'un cartain rang et il existe j tel que $\rho(\rho^j(q)) = \rho^j(q)$. Or, 0 est le seul point fixe de ρ. Ainsi

$$\exists j \geq 0 / \ \rho^j(q) = 0$$

   Comme $\delta(0, \alpha)$ est défini pour tout α, $\delta(\rho^j(q), \alpha)$ est bien défini.

8. Il suffit d'émuler les transitions manquantes. Par exemple,
   - en partant de 3 et par lecture de a, on se replie deux fois (vers 0) puis on avance vers 1 ; on ajoute une transition de 3 vers 1 d'étiquette a
   - en partant de 3 et par lecture de b, on se replie une fois (vers 1) puis on avance vers 2 ; on ajoute une transition de 3 vers 2 d'étiquette b

   On procède de même pour les autres transitions qui manquent et on obtient :



9. *On remarque que depuis n'importe quel état, la lecture de aba mène en 3. Tout mot se terminant par aba est donc reconnu. Réciproquement, on vérifie que toute suite de trois lettres différente de aba mène en un autre état que* 3.
   Le langage reconnu est

$$\Sigma^* aba$$

10. Il faut être attentif pour la matrice de transition. Faire une copie de cette matrice ne suffit pas (la fonction `Array.copy` n'a le bon comportement que pour les tableaux monodimensionnels). Il faut faire une copie de chaque ligne de celle-ci.

```
let copie_afdr a =
   let k = Array.length a.final in
   let t=Array.make k [||] in
   for i=0 to k-1 do
       t.(i) <- Array.copy a.transition.(i)
       done ;
   {final = Array.copy a.final ;
    transition = t ;
    repli = Array.copy a.repli} ;;
```

11. On fait une copie de l'automate à repli. Pour respecter la consigne de complexité, on va remplir la table de transition de cette copie ligne par ligne du haut vers le bas. L'idée est que
    - la ligne numéro 0 est déjà remplie
    - si une case de la ligne $i \geq 1$ n'est pas déjà remplie, sa valeur (via un repli) sera donnée par celle de l'une des cases précédentes.

```
let enleve_repli a =
   let af=copie_afdr a in
```

```
for q=1 to (Array.length a.final)-1 do
  for i=0 to (Array.length a.transition.(0) - 1) do
    if af.transition.(q).(i) = -1 then
      af.transition.(q).(i) <- af.transition.(a.repli.(q)).(i)
    done;
  done;
af;;
```

Dans la double boucle, les opérations sont en temps constant. Il y a $k\lambda$ étapes et donc on a bien une complexité $O(k\lambda)$.

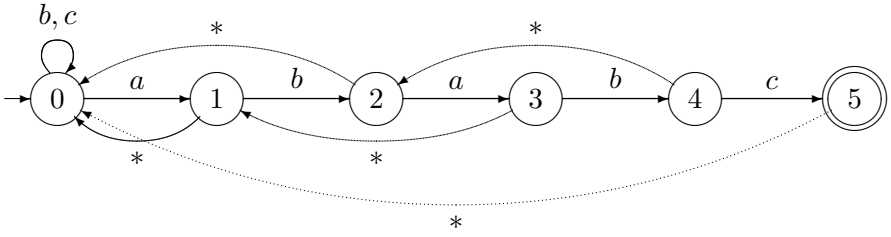
12. On parcourt le mot de gauche à droite ce qui induit un déplacement dans l'automate. Si après lecture de i lettres on tombe sur un état final, on l'ajoute à une liste en construction. Comme on parcourt le mot de gauche à droite, on obtient une liste triée (en ordre croissant ou décroissant selon l'implémentation, voir ci-dessous). La complexité sera $O(n)$ puisque le traitement d'une lettre (lecture d'une transition, test d'état final) se fait en temps constant.
13. J'écris une fonction auxiliaire locale `parcourt : int list → int → int → int list`. Dans l'appel `parcourt v q i`, v est un mot dont on suppose les éléments numérotés à partir de i (et non pas 1 comme usuellement), q est un état. On renvoie la liste des positions dans v où la lecture de v depuis q mène en un état final. Il suffit alors de l'appliquer avec le mot entier depuis l'état 0.

```
let occurrences a u =
   let rec parcourt v q i = match v with
      |[] -> []
      |c::reste -> let d=a.transition.(q).(c) in
                   if a.final.(d) then i::(parcourt reste d (i+1))
                   else parcourt reste d (i+1)
    in parcourt u 0 1;;
```

On fait un unique parcours du mot et chaque lettre est traitée en temps constant. La complexité est $O(n)$ (les caractéristiques de l'automate n'interviennent pas pourvu que l'on ait bien un automate complet).
Comme on ajoute les positions par la gauche depuis le résultat obtenu par récurrence, la liste obtenue est triée par ordre croissant.

3 Automate de Knuth-Morris-Pratt

14. En étiquetant $*$ les transitions de repli, l'automate est



15. L'AFDR $\mathcal{A}_s^{\text{KMP}}$ reconnaît le langage $\Sigma^* s$, c'est à dire l'ensemble des mots dont s est suffixe.
16. *Si $i = 1$, alors $\rho^0(\rho(i-1)) = \rho^0(0) = 0$ par convention. Ainsi $\delta(\rho^0(\rho(i-1)), u_1) = \delta(0, u_1) = 1$. Si l'énoncé était correct, on aurait $\rho(1) = 1$ ce qui est faux ($\rho(1) < 1$ et donc $\rho(1) = 0$). On va donc montrer le résultat uniquement pour $i \geq 2$.*

On se donne $2 \leq i \leq k$ et on cherche le plus grand j tel que $u_1 \dots u_j$ est suffixe de $u_1 \dots u_i$.
Un suffixe de $u_1 \dots u_i$, s'il n'est pas vide, peut être vu comme un suffixe de $u_1 \dots u_{i-1}$ suivi de la lettre u_i. A chaque suffixe non vide de $u_1 \dots u_i$, on associe donc un suffixe de $u_1 \dots u_{i-1}$ qui est suivi de la lettre u_i.
Réciproquement, si on a un suffixe $u_1 \dots u_p$ de $u_1 \dots u_{i-1}$ tel que $u_{p+1} = u_i$, on en déduit un suffixe de $u_1 \dots u_i$.
A moins qu'il n'existe aucun suffixe non vide de u du type $u_1 \dots u_j$, on cherche donc en fait le plus long suffixe de $u_1 \dots u_{i-1}$ du type $u_1 \dots u_p$ et tel que $u_{p+1} = u_i$.

- Le premier candidat est $u_1 \dots u_{\rho(i-1)}$. Il convient si $u_{\rho(i-1)+1} = u_i$. Dans ce cas, $\rho(i) = \rho(i-1) + 1 = \delta(\rho(i-1), u_i)$ et on a la formule demandée.
- Sinon, il faut un $p < \rho(i-1)$. On cherche alors un suffixe de $u_1 \dots u_{i-1}$ du type $u_1 \dots u_p$ avec $p < \rho(i-1)$ et ce sera aussi un suffixe de $u_1 \dots u_{\rho(i-1)}$ (par définition de $\rho(i-1)$). Le second candidat est $u_1 \dots u_{\rho^2(i-1)}$. Il convient si $u_{\rho^2(i-1)+1} = u_i$. Dans ce cas, $\rho(i) = \rho^2(i-1) + 1 = \delta(\rho^2(i-1), u_i)$ et on a la formule demandée.
- Le processus se poursuit avec $\rho^3(i-1)$, etc. jusqu'à trouver le premier $j \geq 1$ tel que $u_{\rho^j(i-1)+1} = u_i$ ou à tomber sur $\rho^j(i-1) = 0$.

Ceci est une preuve un peu informelle qui, je l'espère, a le mérite d'expliquer la formule demandée. Je ne vois pas de preuve rigoureuse vraiment plus convaincante.

17. Le problème principal, selon moi, vient de la structure de données utilisée : on n'a pas accès direct aux éléments d'une liste.
Je vais définir et mettre à jour les tableaux pour remplir les champs `final`, `transition` et `repli`.

- Pour `final`, c'est assez simple car un seul état est terminal.
- Pour `transition`, on introduit une matrice `delta`. La plupart de ses éléments valent -1. Il y a exception pour ceux de la ligne 0 (remplie itérativement) et pour les $\delta(i-1, u_i)$. On utilise une fonction auxiliaire de parcours de liste pour ces dernières (deux arguments : les lettres qui restent à lire et le numéro dans le mot de départ de la première lettre qui reste).
- Pour `repli`, on utilise un tableau `rho`. Là encore, c'est avec une fonction auxiliaire que l'on fait le traitement (mêmes arguments). Comme indiqué en question précédente, il faut particulariser la première lettre (d'où l'appel initial à la fonction auxiliaire avec la queue du mot et l'entier. 2). Dans la fonction auxiliaire, on a une boucle pour "chercher le premier j convenable" (ici le premier état $e = \rho^j(i-1)$ convenable).

```
let automate_kmp s =
 (* le nombre d'\'etat sera k+1 *)
   let k=List.length s in
 (* Initialisation et remplissage pour les \'etats finaux *)
   let fin=Array.make (k+1) false in
   fin.(k) <- true;
 (* Initialisation et remplissage pour les transitions *)
   let delta=Array.make_matrix (k+1) lambda (-1) in
   for c=0 to lambda - 1 do delta.(0).(c) <- 0 done ;
   let rec remplir_delta u i = match u with
      |[] -> ()
      |c::reste -> delta.(i).(c) <- i+1 ;
                   remplir_delta reste (i+1)
   in remplir_delta s 0 ;
 (* Initialisation et remplissage pour les replis *)
   let rho=Array.make (k+1) 0 in
```

```
let rec remplir_rho u i = match u with
   |[] -> ()
   |c::reste -> let e=ref rho.(i-1) in
                while delta.(!e).(c) = -1 do e:=rho.(!e) done;
                rho.(i) <- delta.(!e).(c);
                remplir_rho reste (i+1)
in remplir_rho (List.tl s) 2 ;
{final=fin;transition=delta;repli=rho};;
```

Là encore, il faudrait sans doute plus de commentaires. Notez que l'on pourrait aussi utiliser des références de listes pour obtenir successivement les lettres et utiliser un traitement itératif.

18. On a $\rho(i) = \rho^{j_i}(\rho(i-1)) + 1$. Une récurrence simple montre que (on perd au moins une unité à chaque composition par ρ)

$$\forall q \in \{0, \dots, k\},\ \forall j \leq q,\ \rho^j(q) \leq q - j$$

et ainsi

$$\rho(i) \leq \rho(i-1) + 1 - j_i$$

On en déduit que

$$\sum_{i=1}^{k} j_i \leq \sum_{i=1}^{k} (1 + \rho(i-1) - \rho(i)) = k + \rho(0) - \rho(k) \leq k$$

On a donc bien

$$\sum_{i=1}^{k} j_i = O(k)$$

19. Création et initialisation du tableau des états finaux ont un coût $O(k)$ (y compris le calcul de k).
L'initialisation du tableau des transitions coûte $O(\lambda k)$. Il y a ensuite $\lambda + k$ cases à modifier ce qui garde la complexité $O(\lambda k)$.
L'initialisation du tableau des replis coûte $O(k)$. Pour remplir la case i, on a de l'ordre de j_i opérations. Le remplissage a donc un coût $O(k)$ avec la question précédente.
Finalement, `automate_kmp` a une complexité $O(k\lambda)$.

20. Il suffit de combiner les différentes fonctions.

```
let recherche_kmp s t =
   let a=enleve_repli (automate_kmp s) in
   occurrences a t ;;
```

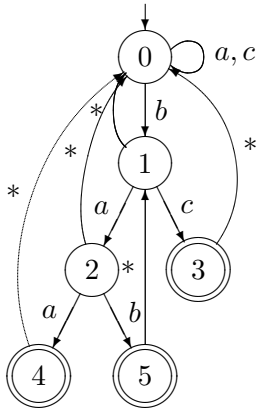
On obtient une complexité globale $O(n + k\lambda)$. Comme $k \ll n$ (la taille du motif est beaucoup plus petite que celle du texte), c'est meilleur que l'approche naïve.
Ici, on imagine que l'on va devoir chercher un motif donné, un bout d'ADN par exemple, dans de nombreux textes (comparaison génétique avec de nombreux sujets). Il est alors intéressant de "prétraiter" le motif : le coût de ce prétraitement est ensuite absorbé par les gains de complexité dans les recherches.

4 Ensemble de motifs et automates à repli arborescents

21. Les mots reconnus sont ceux qui possédnt au moins deux lettres et ne se terminent pas par bb, c'est à dire se terminent par aa ou ab ou ba/ Le langage reconnu est ainsi

$$(a + b)^*(aa + ab + ba)$$

22. On propose l'AFDR suivant (où les replis sont les transitions étiquetées $*$).



23. On calcule les listes pour chaque motif `m` dans `s` et on concatène les listes obtenues. Pour parcourir la liste `s`, on écrit une fonction auxiliaire `int list list` $\rightarrow$ `int list` faisant le travail demandé (le texte `t` est connu de cette fonction).

```
let recherche_dictionnaire_kmp s t =
     let rec ajoute_motif s = match s with
         |[] -> []
         |m::reste -> (recherche_kmp m t)@(ajoute_motif reste)
     in ajoute_motif s;;
```

On a $|S|$ motifs de tailles majorées par k. Chaque recherche pour un motif a un coût $O(n+k\lambda)$. Les recherches ont donc un coût global $O(|S|(n+k\lambda))$. Ceci ne prend pas en compte le coût des concaténations. Le coût de la concaténation `l1 @ l2` est linéaire en la taille de `l1`. Dans notre fonction, les concaténations ont un coût égal à la somme des tailles des différentes listes obtenues et est majoré par $n|S|$. Comme ceci ne change pas la complexité globale, on garde cette version (on pourrait supprimer les concaténations par utilisation d'un accumulateur).

24. *Je ne vois pas comment on peut espérer une démarche même incomplète, en temps limité. On peut juste donner les très grandes lignes de la méthode qui découle de ce qui précède.*

Comme en question 22, on pourrait construire un unique automate à repli de façon arborescente. On commence par créer l'automate sans les replis :

- on part d'un état 0 et crée un nouvel état pour chaque lettre qui est une première lettre d'un des motifs ; on crée aussi les transitions entre 0 et ces nouveaux états (d'étiquette la lettre associé à l'état créé)
- on recommence à partir de chaque nouvel état en ne considérant que les queues des mots qui commencent par la lettre associée à l'état ;
- on poursuit le processus et à la fin on ajoute les transitions entre 0 et lui même.

Il reste à ajouter les replis. Un état q correspond, lors de la création de l'automate, à un préfixe p d'au moins un des motifs de la liste. On cherche le plus grand suffixe p' strict de p qui est aussi préfixe d'un élément de la liste des motifs. p' est aussi associé à un état et on crée un repli de l'état associé à p vers celui associé à p'.
Une première difficulté est de garder un lien entre les états et les préfixes des motifs de la liste. On espère ensuite avoir une formule comme en question 16 pour faire la construction rapide et efficace des replis. C'est la seonde difficulté théorique : vérifier qu'une telle formule est encore valable. C'est certainement ici qu'une hypothèse supplémentaire sur les motifs pourrait s'avérer utile.
Le nombre d'états créés est au maximum égal à $1 + k|S|$. On peut espérer créer les replis en $O(k|S|\lambda)$ et faire la recherche en $O(nk|S|\lambda)$.