RECHERCHE TOLÉRANTE DE MOTIFS DE CHAÎNES

Yves Lepage
ATR-Laboratoires d'Interprétation et Télécommunications
619-0288 Kyouto-hu Soraku-gun Seika-tyou Hikaridai 2-2
lepage@itl.atr.co.jp

RÉSUMÉ

Nous présentons un algorithme pour la recherche tolérante de chaînes plus rapide asymptotiquement en moyenne qu'*agrep* sur la tâche suivante : trouver tous les mots d'un dictionnaire à une distance inférieure à un seuil donné d'un mot donné.

INTRODUCTION

La recherche tolérante est l'opération qui consiste à rechercher les lignes d'un fichier qui contiennent une sous-chaîne à une distance infèrieure ou égale à un seuil k donné d'un motif p (de longueur m). La distance est définie comme le nombre minimal d'insertions, de suppressions ou de substitutions nécessaires à la transformation d'une chaîne en le motif donné.

Par exemple, si on recherche le motif *analogie* avec un seuil de 2, seules les deuxièmes et troisièmes lignes du fichier suivantes seront trouvées.

`analogique`	dist(`analog`, `analogie`) = 2
`explication`	(insertions : `ie`)
`neuroanatomie`	dist(`anatomie`, `analogie`) = 2
	(remplacements : `l` en `t` et `g` en `m`)

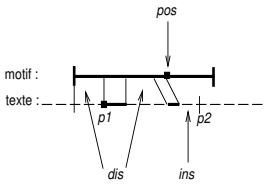
Wu and Manber [Wu & Manber 92] ont proposé récemment une implantation pratique, baptisée *agrep*, de la méthode de Baeza-Yates et Gonnet qui montre un comportement asymptotique en $O(nk\lceil\frac{m}{w}\rceil)$, avec w une constante. *agrep* est considéré comme l'algorithme le plus rapide à ce jour pour ce problème.

1 SOUS-CHAÎNES COMMUNES

Au contraire des méthodes existantes, qui, fondamentalement, améliorent le calcul de la matrice classique de distance, notre méthode recherche la séquence de sous-chaînes la plus longue commune au motif et au texte (les deux problèmes sont équivalents).

Toute séquence candidate peut être définie au moyen de trois entiers seulement : la position du dernier caractère commun au motif (*pos*), la distance passée calculée jusqu'à ce point (somme des maximums des longueurs des parties non-communes en regard du texte et du motif, *dis*), et la longueur du décalage en avant (*ins*).

L'algorithme repose donc sur un tableau contenant des triplets d'entiers. Chaque triplet représente un candidat en cours de construction.



pos	*dis*	*ins*
4	1	0
3	0	1
...	...	...

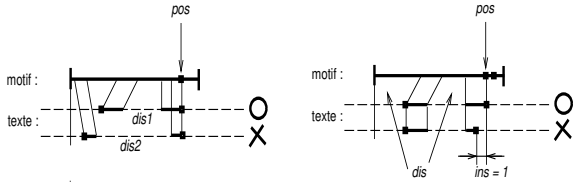
A chaque position donnée du texte, si le caractère suivant dans le texte appartient au motif, un nouveau candidat peut être calculé à partir de tous les candidats compatibles en prenant le minimum sur toutes les distances passées (ligne © dans l'algorithme).

Dans tous les cas, le caractère suivant peut être considéré comme un décalage en avant. Donc, pour chaque candidat, ce décalage, représenté par *ins*, doit être incrémenté (Ⓐ et Ⓑ).

De l'esquisse précédente de l'algorithme, on conclut que l'ensemble des candidats croît sans cesse. Heureusement, des contraintes peuvent être appliquées pour en réduire la taille.

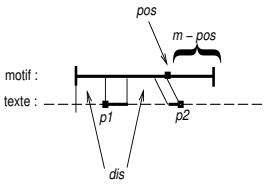
2 CONTRAINTES

Contraintes globales En premier lieu, pour la même position dans le motif et le texte, et sans décalage en avant ($ins = 0$), tout candidat avec une distance passée supérieure peut être éliminé, car il ne pourra plus dans l'avenir retrouver une distance inférieure à celle du candidat ayant la plus petite distance passée. On appelle Min cette contrainte (à gauche ci-dessous, le candidat avec *dis2* est éliminé en faveur de celui avec *dis1*).



En second lieu, l'introduction d'un décalage en avant serait pénalisant si le caractère suivant correspondait au caractère suivant dans le motif. Ces candidats sont donc éliminés. On appelle cette contrainte Continuité : $p[pos + 1] = c$ and $ins = 0$ (à droite ci-dessus, le candidat marqué par un cercle est conservé, alors que celui marqué par une croix n'est pas créé).

Contraintes locales Clairement, on peut éliminer tout candidat pour lequel la somme des distances passées et du décalage en avant dépasse le seuil. C'est la contrainte Candidat : $dis + ins \leq k$.



Enfin, on peut remarquer qu'un candidat peut être déclaré solution avant même d'avoir reconnu le motif en entier. Il suffit que la partie restante du motif soit de longueur inférieure à ce qui reste du seuil : $m - pos < k - dis$. Dès que cette contrainte Solution est remplie, la ligne peut être sortie : elle contient une solution.

3 L'ALGORITHME

Les observations précédentes sont incorporées à l'algorithme comme le montre l'esquisse suivante. Selon que le caractère suivant lu dans le texte appartient ou non au motif, différentes contraintes sont appliquées dans un ordre déterminé afin d'éliminer le plus possible de candidats (Ⓐ, Ⓑ et Ⓒ). Ce n'est que lorsque la contrainte Solution est vérifiée en Ⓒ qu'une ligne peut être sortie.

```
fonction recherche(caractere c)
   si  c ∉ p alors
      pour chaque candidat (pos, dis, ins) faire
Ⓐ        Candidat sur (pos, dis, ins+1)
   sinon -- c ∈ p
      pour chaque candidat (pos, dis, ins) faire
Ⓑ        Continuite/Candidat sur (pos, dis, ins+1)
      pour chaque position π de c dans p faire
Ⓒ        Min/Solution/Candidat sur (π, min_{π>pos} dis, 0)
   fin si
fin fonction recherche
```

4 PERFORMANCES

L'algorithme a été testé sur la tâche suivante : trouver tous les mots d'un dictionnaire de 25 000 mots dans le dictionnaire lui-même, avec tous les seuils possibles (de 1 à la longueur du motif moins 1).

Comme les performances de l'algorithme dépendent du nombre de candidats construits, nous avons mesuré la taille du tableau de candidats lors de l'exécution. Pour cette tâche, la taille maximale a été de 21. En comparaison, la taille moyenne est extrêment basse : 0,41. Ceci explique que cet algorithme soit un concurrent sérieux pour *agrep*.

Bien que, sur l'ensemble des passes de recherche, *agrep* soit plus rapide dans 59% des cas, notre algorithme présente un meilleur comportement asymptotique. Pour un seuil supérieur à 3, *agrep* n'est plus rapide que dans 6% des cas ! La différence Δt des temps d'exécution entre *agrep* et notre algorithme a été mesurée. Il va sans dire que les deux algorithmes rendent exactement les mêmes solutions.

seuil	$k \leq 3$	$k > 3$	total
% de cas	47%	53%	100%
agrep plus rapide	79%	6%	59%
Δt moyen (ms)	-68	95	18
Δt écart-type (ms)	±59	±71	±105

Si on compare les temps d'exécution (en moyenne) des deux algorithmes l'un par rapport à l'autre, par longueurs de motif et seuils, il apparaît clairement que *agrep* n'est plus rapide que pour les petites valeurs. Les cas de petites valeurs étant plus nombreux dans cette tâche, il est normal que *agrep* soit plus rapide dans 59% des cas.

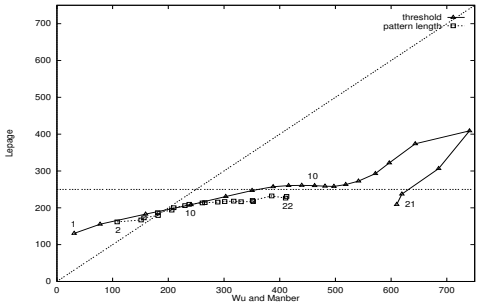


Figure 1: L'algorithme comparé à *agrep*. Temps en moyenne (ms), par longueurs de motif et par seuils.

Le rapport du seuil à la taille du motif montre plus clairement que le comportement de notre algorithme est plus prometteur. Lorsque ce rapport augmente, l'algorithme reste sous la barre des 250 ms, alors que les temps d'exécution d'*agrep* augmentent. Pour des rapports supérieurs à 0,4, notre algorithme est plus rapide que *agrep*.

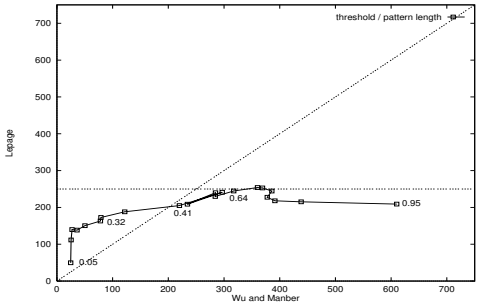


Figure 2: L'algorithme comparé à *agrep*. Temps en moyenne (ms), rapport seuil/taille du motif.

CONCLUSION

Nous avons présenté un algorithme qui peut être un concurrent sérieux d'*agrep*. Il est plus rapide que *agrep* pour des seuils supérieurs à 3, et aussi lorsque le rapport seuil/taille du motif est supérieur à 0,4. Cet algorithme rend possible la recherche de phrases dans des applications de traitement de la langue, puisque, les phrases ordinaires ont souvent plus de dix mots et que les seuils considérés doivent être supérieurs à 5.

BIBLIOGRAPHIE

[Wu & Manber 92] Sun Wu & Udi Manber
Fast Text Searching Allowing Errors
Communications of the ACM, Vol. 35, No. 10, October 1992, pp. 83-91.