Identification de symboles dans des documents déstructurés

Jacques Péré-Laperne*

*1A3i Arcangues 64200 France
Univ. Bordeaux, ESTIA INSTITUTE OF TECHNOLOGY, F-64210 Bidart, France
j.perelaperne@gmail.com,
http://www.1a3i.com

Résumé. Nous décrivons une approche originale pour extraire efficacement les symboles graphiques d'un fichier vectoriel (de type PDF par exemple). Après passage d'un espace d'objets graphiques 2D à une chaîne de codes (1D), l'identification des symboles consiste à rechercher une sous-séquence de codes qui se répète dans le fichier d'entrée. Les travaux de la littérature utilisent l'arbre ou le tableau des suffixes, notre algorithme s'appuie sur le principe du tri par paquets pour identifier les répétitions. La taille et la fréquence sont spécifiées par l'utilisateur.

1 Introduction

L'interprétation d'un fichier vectoriel (PDF, PS, EMF, WFM, IGES, STEP, DXF...), donne une suite ordonnée d'objets graphiques. Dans ce travail nous nous intéressons à l'identification des symboles dans les fichiers vectoriels de type schémas ou plans produits par des logiciels de DAO/CAO (Dessin Assisté par Ordinateur/Conception Assistée par Ordinateur) ou de PAO (Publication Assistée par ordinateur). Ces logiciels dessinent toujours les objets graphiques complexes (symboles) ou simples (rectangles, ellipses) de la même façon, et avec le même ordre chronologique. Partant de ce constat, et des travaux de Péré-Laperne et Couture (2017), ainsi que de ceux de Péré-Laperne (2018) qui précisent comment passer de l'espace des objets graphiques 2D à une chaîne de codes en 1D , sachant que les fonctions de transformation sont insensibles aux translations, aux rotations, ou aux homothéties, l'identification des symboles dans ces fichiers revient à rechercher les sous-chaînes de codes dans ces chaînes de codes issues des fichiers d'entrée. Cette identification des symboles est une des étapes de la méthode (A)KDD (Antropocentric Knowledge Discovery in Database) décrite par Péré-Laperne et Couture (2017) pour la restructuration des documents déstructurés. Cette méthode s'appuie sur les premiers travaux de Fayyad et al. (1996) pour l'extraction des connaissances des données.

Dans la section suivante, nous dressons un bref état de l'art. Dans la section 3, nous faisons le lien entre les répétitions et les paquets de l'algorithme de tri par paquet et nous proposons un algorithme linéaire au niveau du temps et de l'espace d'exécution. Dans la section 4 nous présentons les bénéfices de l'algorithme proposé , avant de conclure dans la dernière section.

2 État de l'art

Dans le domaine des objets graphiques vectoriels, l'état de l'art sur l'identification des symboles est inexistant. Des travaux existent sur l'identification des symboles dans des fichiers images (*bitmap*), mais, les problèmes sont très différents. Ces derniers manipulent des *pixels*, alors que nous utilisons des entités de plus haut niveau (segments, poly lignes, lettres, etc...), qui, même déstructurées, contiennent beaucoup plus d'informations que les *pixels*.

Comme introduit précédement pour identifier les symboles il faut identifier les répétitions. Les travaux de Gusfield (1997), puis ceux de Saha et al. (2008)), pour identifier des répétions dispersées dans l'ADN, se sont heurtés à la taille de la cible génomique. Ces travaux utilisent les arbres des suffixes ce qui entraîne une occupation mémoire (en octets) supérieure à 15 fois la taille du génome. L'utilisation du tableau des suffixes par Franěk et al. (2003) et Narisawa et al. (2007) ramène cette taille à 5 fois la taille du fichier d'entrée (1 fois pour les données en entrée et 4 fois pour le tableau des suffixes). Depuis, les travaux de Puglisi et al. (2010), Yusufu et Yusufu (2015), ou ceux plus récents de Louza et al. (2017) confirment l'abandon de l'arbre des suffixes pour utiliser le tableau des suffixes, pour des raisons d'occupation mémoire. Ces algorithmes nécessitent la création du tableau des suffixes (SA), des plus longs préfixes communs (LCP) et de 2 autres tableaux (BWT,LAST). On constate, cependant, dans les résultats obtenus par Puglisi et al. (2010) (Tab.1), que la somme des temps pour créer les 4 tableaux préparatoires (1,912+0,17,+0,035+0,039) est égal à 144 fois le temps de l'algorithme (0,015) le plus rapide PSY1-1 pour détecter les répétitions.

Temps des traitements (en microsecondes par lettre)									
Fichier	Prétraitement des tableaux					Les algorithmes PSY$_{1-1}PSY_{1-4}$			
	SA	LCP	BWT	LAST		PSY$_{1-1}$	PSY$_{1-2}$	PSY$_{1-3}$	PSY$_{1-4}$
HowTo(*)	1,912	0,178	0,035	0,039		**0,015**	0,017	0,018	0,016
(*) le fichier "Howto" est accessible à l'adresse http ://www.cas.mcmaster.ca/ bill/strings/)									

TAB. 1 – *Extrait des résultats de Puglisi et al. (2010)*

3 Algorithme proposé

Nous partons de la chaîne de départ, de longueur n, notée S : S est une séquence de n codes de l'ensemble Σ, dans notre cas $|\Sigma| \leq 256$. Pour identifier un symbole, dans un schéma, à partir de la chaîne de codes 1D, nous devons trouver les sous-chaînes de codes qui se répètent au moins T_{inf} fois et qui sont composées d'au moins L_{inf} codes. Prenons deux exemples. Le premier, pour identifier les cartouches dans un dossier électrique de 100 folios, le cartouche est présent sur chaque folio, donc si on cherche les sous-chaînes qui se répètent plus de 90 fois (T_{inf}=90) et qui sont composées d'au moins 50 codes (L_{inf}=50 un cartouche est composé d'un très grand nombre de codes voir Fig.1), on est sûr d'avoir identifié les cartouches. Le deuxième pour identifier les protections. Le symbole protection, dans un tel dossier de 100 folios, est présent au moins une trentaine de fois et le nombre de codes qui le compose est

d'une quarantaine (T_{inf}=30 et L_{inf}=40), Donc, si on identifie les sous-chaînes qui se répètent au moins 30 fois, et qui ont une longueur supérieure à 40, on est sûr d'identifier les protections.

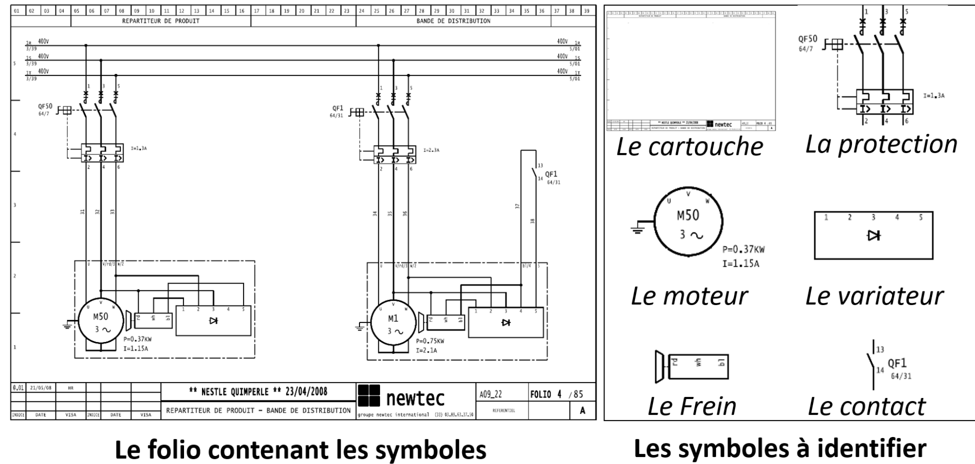

FIG. 1 – *A gauche le folio, à droite les symboles*

Dans les travaux de l'état de l'art, le tableau SA est créé pour obtenir la totalité des suffixes triés, puis le tableau LCP permet de déterminer le nombre de codes identiques entre 2 suffixes triés. Ensuite les autres tableaux servent à identifier les répétitions. Dans ces méthodes, ce qui prend le plus de temps, c'est la création du tableau SA car tous les suffixes sont triés.

Algorithm 1 TriPaquet(R,p)

If $|R| < t$ **then** $insertionSort(Rpd)$ {tri par insertion si nombre suffixe < t}
For $s \in R$ **do** $B[S[p]] := B[S[p]U\{s\}$ {range les suffixes dans les paquets}
For $c \in \Sigma$ **do if** $|B[c]| > 0$ **then** $TriPaquet(B[c], p+1)$ {appel récursif, p incrémenté}

L'algorithme que nous proposons ne trie pas tous les suffixes, ne crée pas le tableau (SA), et, il extrait les répétitions en une seule passe. Notre algorithme repose sur l'algorithme de tri par paquets (bucket sort). L'article de Kärkkäinen et Rantala (2009) décrit ce type de tri qui est linéaire en temps. Pour des raisons de performance Sedgewick et Wayne (2015) montrent qu'il faut passer au tri par insertion, quand le nombre de suffixes du paquet est proche de $\sqrt{|\Sigma|}$. Ils affirme que c'est le problème majeur de presque tous les algorithmes de tri (*Small subarrays are of critical importance in the performance of MSD string sort. We have seen this situation for other recursive sorts quicksort and mergesort*). Donc, si on supprime le tri des paquets les plus petits, on diminue de façon considérable le temps de tri, c'est l'objectif de notre algorithme.

Avant de le décrire, il faut définir la relation qui existe entre les répétitions et les paquets du tri. Une répétition est un ensemble de sous-chaînes répetées. Une répétition dans S est définie par $R_{S,u} = (L_r; i_1, i_2..., i_{T_r})$, où $T_r \geq 2$ et $0 \leq i_1 < i_2... < i_{T_r} \leq n-1$, la longueur de L_r est égale à la longueur des sous-chaînes, la taille T_r est égale au nombre de sous-chaînes

et les i_i sont les adresses des sous-chaînes. Pour l'algorithme de tri, un paquet est égal à la répétition $R_{S,u} = (p+1; i_1, i_2..., i_{T_r})$ ou p est la profondeur de la récursivité et les i_i sont les adresses suffixes du paquet à trier et T_r le nombre de suffixe du paquet. On démontre (assez facilement) que toutes les répétitions correspondent à des paquets et que tous les paquets sont les répétitions de la chaîne S.

Algorithm 2 Répétition(R,p)

```
bProfond := True   {Indicateur du paquet le plus profond de la récursivité est mis à vrai}
For s ∈ R do B[S[p]] := B[S[p]U{s}        {Range les suffixes dans les paquets}
For c ∈ Σ do                               {Pour chaque code}
if (|B[c]| > T_inf) and (p ≤ L_inf) then   {Est ce que le paquet a assez de suffixe ?}
    Repetition(B[c], p + 1)                {Et profondeur < limite alors appel recursif}
if ConditionsPourEnregistrer then          {Si les conditions pour enregistrer sont vrais}
    EnregistreRepetition                   {Alors on enregistre la répétition}
bProfond := False                          {Indicateur du paquet le plus profond est mis à faux}
```

Dans notre algorithme, l'appel de la récursivité n'est exécuté que si le nombre de suffixes du paquet est $\geq T_{inf}$ et si la profondeur de la récusivité est $\leq L_{inf}$. L'indicateur $bProfond$ est mis à vrai en début de procédure, et, remis à faux en fin de procédure. En fin procédure, l'enregistrement des répétitions ne se fait que si $bProfond$ est vrai, le nombre de suffixes du paquet est $\geq T_{inf}$, la profondeur p est $\geq L_{inf}$ et à condition que les suffixes ne se chevauchent pas.

4 Résultats

Seuil	Tri insertion	Pas de tri
256	1,308	0,111
128	1,155	0,123
64	1,112	0,154
32	1,121	0,204
16	1,189	0,285
8	1,298	0,442
Temps en microsecondes par lettre		

TAB. 2 – *Temps des tris par paquets du fichier "HowTo" en fonction d'un seuil*

La colonne "Tri insertion" (Tab.2) est conforme aux résultats de Sedgewick et Wayne (2015), et montre la nécessité de changer de tri quand la taille du paquet est petite. La première colonne contient les valeurs des seuils (c'est à dire le nombre de suffixes du paquet) qui déclenche le changement de tri. La colonne "Pas de tri" correspond au fait que l'on ne fait plus de tri en dessous du seuil indiqué par la première colonne. Là encore, on vérifie très bien les propos de Sedgewick et Wayne (2015). Ce qui prend du temps c'est le tri des petits paquets. Or pour identifier les répétitions il n'est pas nécessaire de les trier.

Algorithme "REPETITION" sur le Fichier "HowTo"						
$T_{inf}(*)$	Longueur des répétitions : L_{inf}					
	8	16	32	64	128	256
256	0,084	0,093	0,095	0,097	0,102	0,103
128	0,087	0,101	0,108	0,115	0,120	0,125
64	0,093	0,123	0,136	0,140	0,145	0,146
32	0,108	0,140	0,161	0,170	0,183	0,194
16	0,115	0,180	0,208	0,227	0,242	0,258
8	0,154	0,245	0,296	0,332	0,356	0,388
(*) Nombre des suffixes dans un paque T_{inf}						

TAB. 3 – *Temps d'éxécution de l'algorithme "REPETITION" sur le fichier "HowTo"*

Dans notre algorithme RÉPÉTITION, le temps pour identifier les répétitions qui ont un nombre de suffixes supérieur à 16 (T_{inf}) et une longueur de sous-chaîne supérieure à 8(L_{inf}) est de 0,115 microsecondes (voir Tab.3). Ce temps est à comparer aux 1,112 microsecondes du tri complet de tous les suffixes (voir Tab.2). Le temps de traitement est divisé par 10. Comparé à ceux de Puglisi et al. (2010) il est 19 fois plus rapide (voir Tab.1). L'espace mémoire occupé par notre algorithme est légèrement supérieur à ceux utilisant les tableaux des suffixes, il faut rajouter l'espace occupé par les compteurs : dans notre cas 1024 octets ainsi que 4 fois la taille du plus grand paquet. On peut diminuer l'espace total occupé à 8 fois la taille du plus grand paquet plus la taille des compteurs, cela complexifie l'algorithme et diminue ses performances d'environ 10%.

La méthode (A)KDD est centrée utilisateur, c'est lui qui fixe les seuils T_{inf} et L_{inf}. La stratégie consiste à identifier en premier les symboles présents le plus grand nombre de fois et/ou les plus grands, puis, ceux qui apparaissent le moins souvent et/ou les plus petits. A chaque itération les symboles identifiés sont supprimés du fichier d'entrée, de cette façon l'itération suivante se fait sur un fichier de taille inférieure, elle est plus rapide.

5 Conclusion

Nous avons proposé un algorithme de détection des répétitions qui est très efficace dans la recherche des répétitions sur des fichiers textuels. Nous avons utilisé des fichiers textuels car c'est le seul moyen de vérifier la performance de l'algorithme par rapport à l'état de l'art. Les premiers résultats obtenus pour la détection des symboles dans les fichiers graphiques sont très encourageants (non décrit dans cet article). Ils feront l'objet de nos prochains travaux. Nous utiliserons les mémoires caches de façon optimale, comme le propose Kärkkäinen et Rantala (2009). Et surtout, nous paralléliserons l'algorithme par paquets, qui se prête très bien à ce type d'optimisation. La recherche des répétitions est l'algorithme principal de la méthode (A)KDD : (Antropocentric) Knowledge Discovery in Database), il est utilisé pour détecter les symboles, mais aussi, les types de segments (pointillés, tirets, axes), les hachures ,les remplissages, les formes simples (cercle, arc, ellipse, rectangle, triangle, carré) et les fonctions.

6 Remerciements

Je tiens à remercier la professeur Nadine Couture pour l'aide apportée dans la structuration de cet article et ses nombreuses relectures.

Références

Fayyad, U. M., G. Piatetsky-Shapiro, et P. Smyth (1996). Advances in knowledge discovery and data mining. pp. 1–34. Menlo Park, CA, USA : American Association for AI.

Franěk, F., W. F. Smyth, et Y. Tang (2003). Computing all repeats using suffix arrays. *J. Autom. Lang. Comb. 8*(4), 579–591.

Gusfield, D. (1997). Algorithms on strings. *Trees and Sequences*, 89–180.

Kärkkäinen, J. et T. Rantala (2009). Engineering radix sort for strings. In A. Amir, A. Turpin, et A. Moffat (Eds.), *String Processing and Information Retrieval*. Springer Berlin Heidelberg.

Louza, F. A., G. P. Telles, S. Hoffmann, et C. D. A. Ciferri (2017). Generalized enhanced suffix array construction in external memory. *Algorithms for Molecular Biology 12*(1), 26.

Narisawa, K., S. Inenaga, H. Bannai, et M. Takeda (2007). Efficient computation of substring equivalence classes with suffix arrays. In *CPM 2007, Proceedings*, pp. 340–351.

Péré-Laperne, J. (2018). (A)KDD for Structuring Destrured Documents. In *Proceedings of the 2018 International Conference on Artificial Intelligence ICAI'18*, Las Vegas, NV. 89, USA.

Péré-Laperne, J. et N. Couture (2017). Restructuring Unstructured Documents. In *SMART INTERFACES 2017*, Venice, Italy, pp. 60–65. Berntzen, L. et al.

Puglisi, S. J., W. F. Smyth, et M. Yusufu (2010). Fast, practical algorithms for computing all the repeats in a string. *Mathematics in Computer Science 3*(4), 373–389.

Saha, S., S. Bridges, Z. V. Magbanua, et D. G. Peterson (2008). Empirical comparison of ab initio repeat finding programs. *Nucleic Acids Research 36*(7), 2284–2294.

Sedgewick, R. et K. Wayne (2015). *Algorithms, Fourth Edition (Deluxe) : Book and 24-Part Lecture Series* (1st ed.). Addison-Wesley Professional.

Yusufu, M. et G. Yusufu (2015). Efficient Algorithm for Extracting Complete Repeats from Biological Sequences. *International Journal of Computer Applications 128*(16), 33–37.

Summary

We describe an approach to efficiently extract graphical symbols from a vector file (such as PDF). After passing from a space of 2D graphic objects to a code string (1D), the identification of the symbols consists in looking for a repeated sub-sequence of codes in the input file. The works of the literature use the tree or array of suffixes. Our algorithm is based on the bucket sort algorithm in order to identify repetitions. The size and frequency, are specified by the end-user.