Apprentissage pour la Recherche d'Information textuelle et multimédia : Construction d'Arbres de Suffixes

Cédric HERPSON

31 mars 2008

Table des matières

Introduction

Travailler sur de grandes chaines de caractères et traiter celles-ci afin d'en extraire les motifs récurrents par correspondance exacte ou approchée est un travail aujourd'hui courant pour les chercheurs en Sciences de la Vie ou en fouille de données. Si l'on ne considère que la base de données GenBank[1], la taille et le nombre des séquences qui y sont stockées doubles tous les 16 mois. Cette augmentation nécessite par conséquent le développement de méthodes à même de répondre rapidement à de nombreuses requêtes sur des séquences toujours plus grandes, éventuellement distantes.

Les arbres de suffixes sont une structure de données à même d'offrir une réponse rapide à des requetes s'apparentant à la recherche de patrons au sein d'une chaine de caractères. Ainsi, la recherche d'une correspondance exacte entre une chaine de caractères fournie en entrée et les sous-chaine d'une chaine de données peut être résolue en un temps proportionnel à la longeur de la chaine d'entrée. Cependant, le temps nécessaire à la construction d'un arbre de suffixes limitait jusqu'à présent l'utilisation de cette structure, les meilleurs algorithmes nécessitant 1h30 pour construire l'arbre de suffixes associé à 1 chromosome du génome humain.
Si les premières chaines sur lesquelles les recherches ont porté pouvaient être intégralement stockées dans la mémoire vive des ordinateurs, les volumes de données actuels ne le permettent plus. Les traitements effectués nécessitent aujourd'hui le stockage d'une partie des informations sur le disque dur, mémoire lente, synonyme d'une chute des performances, que les algorithmes développés jusqu'à présent ne permettaient pas de compenser.

Nous rappellerons dans une première partie la définition d'un arbre de suffixes ainsi que les grands principes liés à la gestion et au coût des accès mémoire sur une architecture moderne. Nous introduirons dans une seconde partie les algorithmes de Ukkonen et Hunt, les plus efficaces dans leurs domaines respectifs, avant de présenter l'algorithme TDD et les avantages de cette dernière approche.

[1] Gérée par le Centre National pour l'Information Biotechnologique américain dans le cadre d'une collaboration internationale (INSDC)

Chapitre 1

Notions élémentaires

1.1 Arbre de suffixes

Un arbre de suffixes est une structure de données qui permet de représenter exactement tous les suffixes d'un texte.

L'interêt de cette structure est qu'elle permet d'effectuer une recherche de motif dans une chaine de caractères dans un temps qui dépend uniquement de la longueur du motif et non du texte dans lequel s'éffectue la recherche.Celle-ci se réalise en effet par la lecture du motif recherché et par le parcours simultané de l'arbre avec les lettres lues.

Cette structure permet d'effectuer :
- La recherche d'un motif exact
- La recherche d'un motif approché (x erreurs autorisées)
- La recherche de répétitions
- La recherche de séquences cycliques
- La recherche du plus long facteur commun entre 2 chaines de données

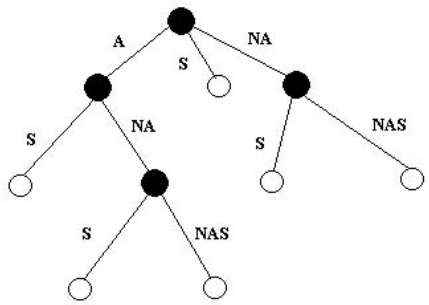

Fig. 1.1 – Exemple d'arbre de suffixes : Ananas

Lors de la recherche d'un motif exact, plusieurs cas peuvent survenir :

1. On ne peut pas descendre dans l'arbre par la lettre lue, il n'y a pas d'occurrence du motif recherché dans le texte.
2. Une fois le motif lu, on se retrouve sur une feuille. Il n'y a donc qu'une occurence du motif dans la chaine, la position de celle-ci peut être obtenue par le numéro de la feuille.
3. Une fois le motif lu, on se retrouve sur un noeud non terminal. Dans ce cas, le nombre de feuilles présentes dans le sous arbre du noeud courant correspond au nombre d'occurences du motif dans le texte.

1.2 Mémoires et Caches

Sur les architectures modernes, le système gère de manière transparente différents types de mémoire dont la taille et les temps de réponse varient de manière significative. Afin de bien comprendre les limites des algorithmes existants et les avantages de l'approche proposée par Sandeep Tata & Al, il nous a semblé nécessaire d'effectuer un rapide rappel sur la pagination à la demande et le fonctionnement des caches.

1.2.1 Principe général de la pagination à la demande

Une application n'a, à un instant donné, besoin que d'un sous ensemble de ses informations. Plutôt que de charger l'intégralité du programme et de ses données en mémoire, va-et-vient global couteux en temps et en espace, le systeme découpe les programmes en morceaux de taille fixe (les pages) et ne charge dans la mémoire physique que celles qui sont référencées à l'instant courant. Le système éffectue donc un va-et-vient au niveau des pages en fonction des besoins.

D'autre part, les temps d'accès à la mémoire vive et au disque dur étant trop importants au regard des cycles processeur, l'utilisation de différents niveaux de cache entre celui-ci et la mémoire vive s'est généralisée. Ces caches, d'une taille équivalente à quelques pages (plusieurs dizaines de kilo-octets) ayant en effet des temps de réponse beaucoup plus proches de la cadence des processeurs.

1.2.2 Principe général du mécanisme de cache

Soient deux niveaux contigus de la hiérarchie mémoire : mémoire rapide et mémoire lente. Ces deux niveaux sont respectivement joués par le cache et la mémoire vive et/ou par la mémoire vive et le disque dur.

Accès en lecture : L' adressage se fait sur la mémoire lente, mais les accès sont toujours réalisés sur la mémoire rapide. Lors des accès, si l'information voulue n'est pas présente dans la mémoire rapide il y a défaut de cache. Il faut alors la transferer de la mémoire lente vers la mémoire rapide (chargement).

Accès en écriture : L'écriture s'effectue dans la mémoire rapide, on recopie ensuite l'information en mémoire lente, éventuellement de maniere différée. Il y a un défaut de cache quand la mémoire rapide est pleine, il faut alors enlever préalablement à toute écriture une partie de l'information présente en mémoire rapide en essayant de maintenir les informations les plus "utiles" (politique de remplacement).

Afin de minimiser le nombre de défauts de page et donc le nombre d'accès en lecture/écriture à la mémoire lente, les politiques de chargement et de remplacement des pages tiennent compte de deux principes fondamentaux, la localité spatiale et la localité temporelle.

Localité spatiale : Si un élément x est référencé à un instant t, les emplacements voisins ont de fortes probabilités d'être référencés dans un futur proche.

- accès(x, t) $\Rightarrow$ probabilité forte d'accés $(x + d, t + \epsilon)$

Localité temporelle : Un élément x référencé à un instant donné a une forte probabilité d'être à nouveau référencé dans un futur proche.

- accès(x, t) $\Rightarrow$ probabilité forte d'accés $(x, t + \epsilon)$

Dans le cas d'une politique LRU (Least Recently Used), on utilise ainsi la propriété de localité temporelle, la page victime est celle dont la dernière référence est la plus ancienne.

Le rapport entre le nombre d'accès provoquant un défaut et le nombre total d'accès est une manière courante de mesurer l'efficacité du cache et la bonne exploitation de celui-ci par les programmes.

Chapitre 2

Principaux algorithmes existants

Les algorithmes existants s'organisent en 2 classes : ceux dont la taille du problème permet de stocker l'intégralité des données source ainsi que l'arbre en mémoire vive, et ceux qui traitent des volumes de données nécessitant des accès disque. Dans ces domaines, les algorithmes considérés jusqu'à présent comme les plus éfficaces étaient respectivement les algorithmes de Ukkonen et de Hunt.

2.1 Structure stockée en mémoire vive : Ukkonen

L'algorithme de Ukkonen base la construction de l'arbre sur le principe qu'il est possible de compléter itérativement un arbre des suffixes partiellement construit. Il accède séquentiellement à la chaine de caractères source et effectue une mise à jour de l'arbre des suffixes par des accès "aléatoires" (au sens de la localité spatiale).
La lecture de la chaine s'éffectue de la gauche vers la droite. A la lecture du $i+1^{iéme}$ caractère , on crée une nouvelle branche à la racine si celui-ci n'a jamais été rencontré et l'on ajoute à tous les précédents suffixes dont le caractère est identique au caractère de rang i une nouvelle branche associée du caractère i+1 de la séquence lue.
Afin de ne pas avoir à parcourir l'arbre pour trouver les noeuds auxquels ajouter une branche, on maintient à chaque itération une liste de pointeurs permettant d'accéder directement à leur position dans l'arbre. Ces références vers les noeuds où vont s'ajouter les futures branches de l'arbres permettent d'obtenir un algorithme en O(n) (avec n la taille de la séquence).

Les pointeurs permettant de traverser l'arbre sans éffectuer de recherche sont représentés en rouge sur la figure 2.1 ci-après[1].

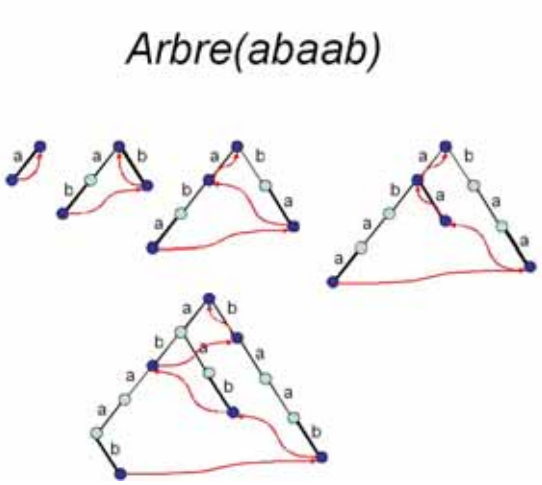

FIG. 2.1 – Construction d'un arbre des suffixes

[1] Extrait de la présentation d'Esko Ukkonen à l'Erice School(2005)

Si les références vers les noeuds où s'ajouteront les futures branches permettent bien d'obtenir un algorithme en O(n), cet algorithme ne tiens pas compte des caractéristiques liées à la gestion des accès mémoire par les ordinateurs.
En effet, comme introduit en 1.2, les architectures actuelles utilisent la pagination à la demande et les principes de localité spatiale et temporelle pour accroitre les performances des applications. Le fait de "sauter" à chaque itération d'une position dans l'arbre à une autre de maniére "aléatoire" (du point du vue du systeme) va entrainer des défauts de page et de cache à répétition. Chaque défaut nécessitant le chargement d'une page de la mémoire physique vers le cache voire, du disque dur vers la mémoire, les performance de cet algorithme sont donc fortement impactées par les va-et-vient effectués par l'Algorithme de Remplacement de Page (ARP) et par les politiques de remplacement implémentés au niveau des caches et du TLB.

2.2 Structure stockée sur le disque dur : Hunt

L'approche développée par Hunt & Al abandonne l'idée de conserver un algorithme de complexité O(n) proposée par Esko Ukkonen. En effet, si la (très) faible localité spatiale de ce dernier peut être acceptable lorsque les données sont stockées en mémoire vive, les temps d'accès dans le cas d'un stockage sur disque magnétique ne le permettent plus.
De même, et afin de ne pas avoir à modifier à l'itération i+1 une portion d'arbre des suffixes préalablement construite et stockée sur le disque, le choix de ne plus éffectuer la création de l'arbre des suffixes en une "passe" a également été fait. Ces choix permettent de construire de maniére indépendante les sous-arbres et de ne conserver ainsi en mémoire que la partition en construction de l'arbre des suffixes.

Le nombre de partitions et la longeur des préfixes utilisés pour la construction de l'arbre des suffixes sont déterminés par l'algorithme en fonction de la taille de la mémoire vive allouée (qui sert de borne supérieure à la taille d'une partition) et de l'espace disque disponible.

L'algorithme avancé par Ela Hunt & Al a la propriété d'assurer une meilleure localité spatiale que l'algorithme de Ukkonen de par l'abandon des liens traversant l'arbre des suffixes. Cependant, le fait de devoir parcourir de manière répétitive l'arbre stocké sur disque pendant la construction de l'arbre des suffixes crée un goulot d'étranglement qui va borner de manière incomprésible les temps de réponse de l'algorithme par la vitesse de lecture/écriture sur disque dur.

Avec pour objectif de s'affranchir, au moins en partie, des limitations des 2 algoritmmes précédents en alliant rapidité de calcul et taille des données traitées, Tata, Hankins et Patel ont introduit en 2004 une nouvelle technique de construction d'arbre basée sur une représentation des données sur le disque : TDD (Top-Down Disk-Based technique).

Chapitre 3

Top-Down Disk-Based technique

La solution proposée est composée d'un algorithme de construction d'arbre de suffixe de complexité $O(n^2)$, à priori plus couteux que l'approche de Ukkonen en $O(n)$, couplé à une stratégie de gestion des accès mémoire basée sur une optimisation des politiques associées aux différents types de caches.

3.1 Partition and Write Only Top Down

L'algorithme de construction d'arbre de suffixes PWOTD dont le pseudo-code est présenté sur la figure 3.1 de la page 6 est lui même constitué de deux étape. La première vise à partitionner la chaine fournie en entrée afin de préparer la construction de sous-arbres indépendants effectuée dans la seconde partie.
Le nombre de parties (et donc de sous-arbres) est déterminé en fonction du nombre de mots de l'alphabet (A) et de la longueur des préfixes souhaitée (prefixlen) : $|A|^{prefixlen}$.

Afin de visualiser les différentes étapes de l'algorithme, nous nous appuierons dans cette partie sur la construction de l'arbre des suffixes de la chaine ATTAGTACA$ ($: caractère terminal). L'alphabet est ici de taille 4 et l'on fixe la taille de la fenêtre des préfixes à 1.

A	T	T	A	G	T	A	C	A	$
0	1	2	3	4	5	6	7	8	9

3.1.1 Partition

Dans la mesure où prefixlen vaut 1, le nombre de parties est égal à la taille de l'alphabet. L'algorithme scrute alors la chaine initiale afin d'associer à chaque préfixe (ici A,T,G,C) les coordonnées des suffixes [0..8] détectés.

$$\text{A} \Longrightarrow \begin{cases} \text{ATTAGTACA\$} \\ \text{AGTACA\$} \\ \text{ACA\$} \\ \text{A\$} \end{cases} \qquad \text{A} \Longleftrightarrow \{0,3,6,8\} \qquad \text{T} \Longrightarrow \begin{cases} \text{TTAGTACA\$} \\ \text{TAGTACA\$} \\ \text{TACA\$} \end{cases} \qquad \text{T} \Longleftrightarrow \{1,2,5\}$$

$$\text{G} \Longrightarrow \text{GTACA\$} \qquad \text{G} \Longleftrightarrow \{4\} \qquad \text{C} \Longrightarrow \text{CA\$} \qquad \text{C} \Longleftrightarrow \{7\}$$

Une fois la partition effectuée, l'algorithme PWOTD peut entamer la deuxième partie de son traitement. Celui-ci consiste en la construction séquentielle de l'arbre des suffixes en effectuant pour chaque ensemble de suffixes obtenu en phase 1° un traitement indépendant.

Algorithm PWOTD(*String*,*prefixlen*)
Phase1:
Scan the *String* and partition *Suffixes* based
on the first *prefixlen* symbols of each suffix
Phase2: Do for each partition:
1. START BuildSuffixTree
2. Populate *Suffixes* from current partition
3. Sort *Suffixes* on first symbol using *Temp*
4. Output branching and leaf nodes to the *Tree*
5. Push the nodes pointing to an unevaluated range
   onto the *Stack*

While *Stack* is not empty
   6. Pop a node
   7. Find the Longest Common Prefix (LCP) of
      all the suffixes in this range by checking
      the *String*
   8. Sort the range in *Suffixes* on the first
      symbol using *Temp*
   9. Write out branching nodes or leaf nodes to *Tree*
   10. Push the nodes pointing to an unevaluated range
       onto the *Stack*

11. END

FIG. 3.1 – Algorithme de construction d'arbre de suffixes

3.1.2 Construction de l'arbre des suffixes

La construction de l'arbre des suffixes nécessite l'utilisation de 4 structures de données différentes :
- La chaine fournie en entrée stockée sous forme d'un tableau
- Un tableau de suffixes (*Suffix*)
- Une structure temporaire utilisée pour le tri des suffixes (*Temp*)
- L'arbre des suffixes en construction

Pour chaque ensemble de suffixe obtenu lors de la première phase, l'algorithme va supprimer les "prefixlen" premiers caractères ayant servis au partitionnement (et donc communs au sein de chaque ensemble), stocker les suffixes tronqués dans *Suffix* et mettre à jour les références vers la chaine initiale.
L'exemple du traitement ainsi décrit est présenté pour le caractère "T" ci-dessous.

$$\mathrm{T} \Longrightarrow \begin{cases} \text{TTAGTACA\$} \\ \text{TAGTACA\$} \\ \text{TACA\$} \end{cases} \qquad \text{Suppression du prefixe "T"} \qquad \mathrm{T} \Longrightarrow \begin{cases} \text{TAGTACA\$} \\ \text{AGTACA\$} \\ \text{ACA\$} \end{cases}$$

$$\mathrm{T} \Longleftrightarrow \{1,2,5\} \qquad\qquad \mathrm{T} \Longleftrightarrow \{2,3,6\}$$

Les chaines sont alors copiées dans *Temp* afin d'être traitées par l'algorithme "count-sort" qui trie celles-ci selon leur premier caractère en comptant simultanément le nombre de fois où ce caractére apparait en tête d'un suffixe.

- Les caractères dont le nombre d'occurence est de 1 correspondent alors à des feuilles de l'arbre des suffixes.
- Les caratères dont le nombre d'occurence est supérieur correspondent à des noeuds de l'arbre qu'il va falloir déplier.

A cette étape, les branches totalement déterminées sont transcrites dans l'arbre des suffixes. Si certains noeuds ne sont pas dépliés, l'algorithme va allouer de la place dans l'arbre des suffixes en construction puis effectuer une recherche du plus long suffixe commun au sein de chaque ensemble de suffixe associé à un noeud. Le préfixe obtenu servira de branche dans l'arbre en construction.

L'algorithme peut alors réexécuter les étapes précédentes en développant récursivement les différents sous-arbres jusqu'à aboutir sur des feuilles.

Dans le cas des suffixes du caractère "T", la lettre "A" apparait 2 fois et le "T" 1 fois. Le suffixe TAGTACA$ peux donc être directement inséré dans l'arbre tandis qu'une nouvelle itération s'avère nécéssaire pour la lettre "A".

$$A \Longrightarrow \begin{cases} AGTACA\$ \\ ACA\$ \end{cases} \quad \text{Suppression du } \underline{\text{plus long préfixe commun}} \text{ "A"} \quad A \Longrightarrow \begin{cases} GTACA\$ \\ CA\$ \end{cases}$$

Dans l'exemple, les groupes "G" et "C" ne comportent qu'un seul élément, l'algorithme s'arrete donc là pour la construction du sous-arbre issu de la lettre "T".

Une fois le sous-arbre entiérement connu, celui-ci est recopié sur le disque dur et l'algorithme réitére alors le processus sur le ou les autres ensembles obtenus à l'issu de la phase 1. L'arbre des suffixes finalement obtenu est représenté sur la figure 3.2

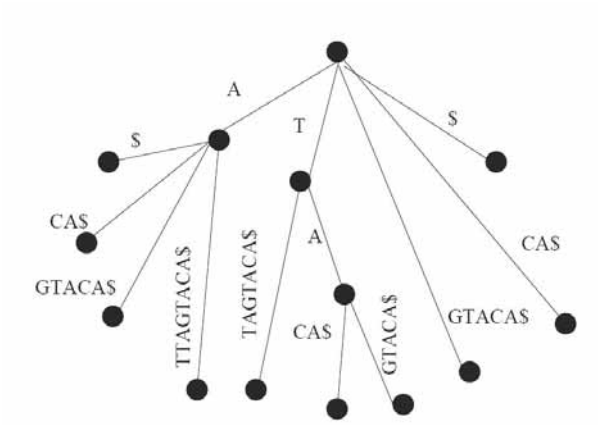

FIG. 3.2 – Arbre des suffixes de la chaine ATTAGTACA$

3.2 Politique de gestion de la mémoire

Les performances des architectures disposant de cache[1] et les politiques de gestion de ces derniers sont fortement dépendantes des programmes qui les utilisent. Afin d'exploiter au mieux les propriétés de leur algorithme, Tata & Al ont donc pris en considération la taille et le type d'utilisation de chacune des 4 structures servant à la construction de l'arbre des suffixes afin de déterminer pour chacune d'elle la politique optimale.

Le plus gros volume de données manipulé est celui de l'arbre des suffixes, celui-ci, une fois complet, occupant un espace plus de 10 fois supérieur à celui de la chaine initiale. Du fait de l'indépendance entre les sous-arbres de l'arbre présentée en 3.1, la majorité des accès en écriture dans cette structure sont faits séquentiellement.
La structure de données représentant l'arbre des suffixes possède donc de bonnes propriétés de localité spatiale et temporelle. Le choix de ne conserver en mémoire rapide que les dernières pages accédées au sein de l'arbre est donc particulièrement adaptée, permettant ainsi de ne pas surcharger la mémoire avec des portions d'arbre qui ne seront accédées que pendant une période bien délimitée. Une politique de remplacement des pages LRU a donc été retenue par les auteurs.

Du fait de la structure de l'algorithme, la réalisation du tri et la recopie des différents suffixes une fois celui-ci réalisé risque d'engendrer de nombreux défauts de cache.

[1] toutes à l'exception des systèmes temps réel dur qui ne tolèrent pas l'indéterminisme lié à leur utilisation.

L'ordre dans lequel vont être triés les suffixes ne peut en effet être anticipé. Aussi, les accès mémoire liés au déplacement de ces derniers sont, du point de vue du système, aléatoires. L'espace mémoire alloué au tableau de suffixe est donc relativement important afin de limiter autant que possible les va-et-vient mémoire rapide/mémoire lente. Du fait de la taille réservée, les auteurs ont considéré qu'une politique LRU pourrait également convenir.

La chaine de caractères contenant les données sources est quant-à-elle accédée sur toute sa longueur durant toute la durée de la construction de l'arbre des suffixes. Aussi, les accès à cette chaine sont ceux qui vont engendrer le plus de défauts de cache.
Une telle constatation, synonyme d'un ralentissement du systeme, pourrait inviter l'utilisateur à réserver le plus grand emplacement mémoire disponible aux données. Les mesures de performances effectuées par les auteurs semblent indiquer que le gain obtenu par page supplémentaire allouée aux données n'est pas satisfaisant et qu'une utilisation efficiente de la mémoire passe par l'allocation de pages supplémentaires au tableau de suffixes.

3.3 Comparatifs des résultats obtenus

L'approche utilisée par l'algorithme TDD permet d'améliorer significativement les temps de construction des arbres de suffixes, que ceux ci-soient, ou non, stockés en mémoire vive.

Le gain de performance obtenu lors des comparaisons avec l'algorithme de Ukkonen représente une division du temps de réponse ce situant entre un facteur 2 et un facteur 10. L' algorithme TDD accroit son avance sur les ensemble de test composés du nombre le plus important de symboles ($guten95 < unif4 < dmelano < swp20 < unif40$) .

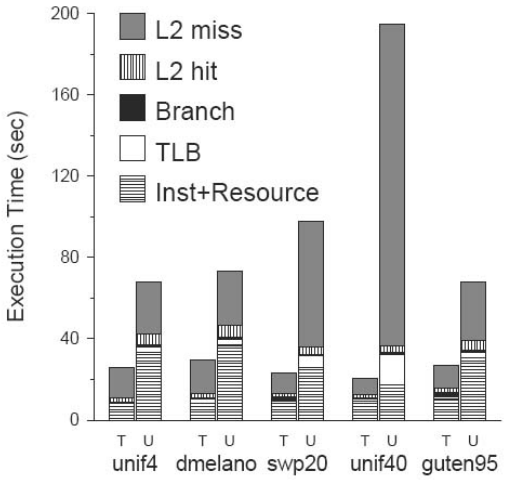

Fig. 3.3 – Comparaison Tdd (T) - Ukkonen (U)

Il est à noter que lors de ces comparaisons, toutes les structures résidant en mémoire, les optimisations liées à la gestion des caches n'interviennent pas ; seul l'algorithme PWOTD, d'une complexité théorique dans le pire cas en $O(n^2)$, est donc considéré.
La meilleure gestion de la localité compense ainsi la plus forte complexité initiale de l'algorithme. Comme on peut le voir sur le graphique 3.3, le temps passé dans des opérations de rechargement suite à un défaut de cache L2 est la source la plus importante du retard de l'algorithme Ukkonen.

Les résultats obtenus lors de la comparaison des temps de réponse sur des structures résidant non plus uniquement en mémoire mais également sur disque entre l'algorithme TDD et l'algorithme de Hunt valident également les choix effectués. Le fait que l'algorithme de Hunt requiert d'accéder réguliérement aux données stockées sur le dique pour construire l'arbre semble la principale raison des écarts obtenus.

Data Source	**Symbols** (10^6)	**Hunt (min)**	**TDD (min)**	**Speedup**
swp	53	13.95	2.78	5.0
H.Chr1-50	50	11.47	2.02	5.7
guten03	58	22.5	6.03	3.7
trembl	338	236.7	32.00	7.4
H.Chr1	227	97.50	17.83	5.5
guten	407	463.3	46.67	9.9
HG	3,000	—	30hrs	—

FIG. 3.4 – Comparaison Tdd - Hunt

Ainsi, comme le montrent les résultats de la figure 3.4, TDD offre des temps de réponse significativements plus rapides (de 5 à 10 fois), permettant par exemple la construction de l'arbre des suffixes d'une chaine de 3.10^9 symboles (le génome humain) en moins de 30 heures sur une machine monoprocesseur (contre plus de 69 heures auparavant).

Conclusion

La construction de l'arbre par l'algorithme PWOTD, bien que d'une complexité théorique supérieure à celle réalisée par l'algorithme de Ukkonen & Al, bénéficie de nombreux avantages. La prise en compte des contraintes matérielles dans la réflexion menant au développement de l'algorithme à ainsi permis de prendre conscience de l'importance de ces facteurs.
Le fait de ne plus utiliser de références inter-branches et d'utiliser une gestion des caches spécifique à la structure de l'algorithme permet en effet de profiter d'une plus forte localité spatiale, offrant ainsi de meilleurs temps de réponse que les algorithmes existants, et ce quelle que soit la taille de l'arbre et le support de stockage utilisé.

On pourrait remarquer que la mise en oeuvre d'une politique de remplacement des pages aussi complexe que LRU nécessite le tri continuel des différentes pages selon leur date d'accès. L'utilisation d'une politique sous-optimale comme l'algorithme de l'horloge pourrait abaisser le coût de gestion en temps processeur des buffers sans grèver de manière significative l'efficacité de ces derniers, accroissant ainsi les performances.

D'autre part, l'indépendance des différents sous-arbres créés lors de la phase de partitionnement permet d'envisager, pour de gros volumes de données, une distribution des calculs permettant d'accroitre les performances de l'algorithme de maniére autrement plus conséquente. En fonction de la taille des séquences traitées, le développement d'une telle architecture pourrait amener à la généralisation de l'utilisation des arbres de suffixes.

Bibliographie

Declarative Querying For Biological Sequences : Sandeep Tata - University of Michigan (2007)

Practical Suffix Tree Construction : Sandeep Tata - Richard A.Hankins - Jignesh M.Patel - University of Michigan (2004)

Efficient implementation of lazy suffix trees : R. Giegerich - S. Kurtz - J. Stoye - University of Bielefeld, Germany (2003)

Suffix Tree Construction : Arthur Dardia - Suzanne Matthews - Polytechnic Institute, New York (2005)

Suffix tree and suffix array techniques for pattern analysis in strings : Esko Ukkonen - Univ Helsinki, Erice School (Oct 2005)

On-line construction of suffix trees : Esko Ukkonen (2004)

A Database Index to Large Biological Sequences : Ela Hunt - Malcolm P.Atkinson - Robert W.Irving - University of Glasgow (2001)