SUFFIX TREE DAN SUFFIX ARRAY

Alwi Alfiansyah Ramdan – 135 08 099

Program Studi Teknik Informatika
Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
e-mail: if18099@students.if.itb.ac.id

ABSTRAK

Makalah ini membahas tentang pohon, pengembangan pohon untuk dapat melakukan berbagai proses pada string secara efektif dan efisien, serta pengembangan pohon tersebut menjadi sebuah struktur data baru berupa sebuah larik/array yang dapat melakukan berbagai fungsi yang dapat dilakukan oleh pohon tersebut di atas dengan lebih ringkas dan efisien. Pohon digunakan dalam berbagai bidang karena fungsinya yang dapat diimplementasikan dengan mudah di berbagai bidang tersebut. Salah satu kegunaan pohon tersebut adalah melakukan proses-proses yang melibatkan string sebagai objeknya, yang disebut dengan *Suffix Tree*. Proses-proses tersebut misalnya adalah pencarian substring, mencari berbagai infrmasi yang berkaitan dengan string tersebut, mencari palindrom dalam string dan mencari *the longest common prefix* pada string. Namun ternyata ada struktur data hasil pengembangan *suffix tree* tersebut yang dapat melakukan beberap proses tersebut di atas lebih cepat dan efektif, yaitu *Suffix Array*.

Kata kunci: pohon, string, *suffix tree*, *suffix array*.

1. PENDAHULUAN

Dewasa ini, informasi berjalan begitu cepat. Berbeda dengan masa lalu, saat ini barang siapa yang mendapatkan informasi terlebih dahulu, maka peluang orang tersebut untuk sukses jauh lebih besar dari pada orang lain yang terlambat mendapatkan informasi. Banyak cara mendapatkan suatu informasi, salah satunya dengan melakukan pencarian. Pencarian pun bisa berbagai macam, ada pencarian file dan ada juga pencarian teks. Ada berbagai macam cara pencarian teks, salah satunya adalah dengan menggunakan *binary seacrh* dengan pohon.

Akhir-akhir ini, telah dkembangkan suatu teknik baru dalam pencarian atau pengolahan lainnya seputar teks menggunakan pohon. Teknik algoritma baru tersebut adalah *Suffix Tree*. Setelah *suffix tree* tersebut, ditemukan pula *Sufix Array* yang merupakan perkembangan dari *suffix tree*.

2. POHON[2]

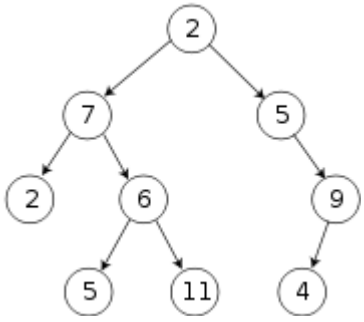
Dalam kehidupan sehari-hari, orang telah lama menggunakan pohon untuk menggambarkan hirarkhi. Misalnya pohon silsilah keluarga, struktur organisasi, dan organisasi pertandingan. Para ahli bahasa menggunakan pohon untuk menguraikan kalimat, yang disebut pohon parsing (*parse tree*).

2.1 Definisi Pohon

Pohon merupakan salah satu penerapan teori graf. Seperti yang telah dijelaskan sebelumnya, konsep pohon sangat penting dalam berbagai bidang, misalnya dunia informasi.

Pohon merupakan suatu bentuk graf yang memiliki sifat-sifat khusus. Definisi pohon adalah sebagai berikut.

Pohon adalah graf tak-berarah terhubung yang tidak mengandung sirkuit.



Gambar 1. Contoh pohon

2.2 Pohon Berakar

Pohon berakar (*rooted tree*) adalah pohon yang sebuah simplunya diperlakukan sebagai akar dan sisi-sisinya diberi arah sehingga menjadi graf berarah. Pada makalah ini, arah pada sisi dapat dibuang, karena setiap simpul di pohon harus icapai dari akar, maka lintasan di dalam pohon berakar selalu dari atas ke bawah.

2.3 Beberapa Terminologi Pohon Berakar

Anak dan Orang Tua

Pada contoh pohon di atas, simpul 7 dan 5 adalah anak dari simpul 2, dan simpul 2 adalah orang tua dari simpul 7 dan 5.

Lintasan

Lintasan dari simpul v_1 ke simpul v_k adalah untutan simpul-simpul $v_1,v_2,v_3,...,v_k$ sedemikian sehingga v_i adalah orang tua dari v_{i+1} untuk $1 \leq i < k$. Pada Gambar 1, lintasan dari 7 ke 11 adalah 7, 6, 11. Panjang lintasan adalah jumlah sisi yang dilalui dalam suatu lintasan, yaitu k-1. Panjang lintasan dari 7 ke 11 adalah 2.

Upapohon / Subpohon

Jika simpul selain akar dijadikan akar baru, maka simpul-simpul keturunan hinggadaun di bawahnya merupakan upapohon dari pohon tersebut.

Derajat

Pada graf, derajat menunjukkan banyaknya sisi yang bersisian dengan simpul yang ingin ditunjukan derajatnya. Pada pohon, derajat adalah banyaknya anak pada suatu simpul.

Daun

Daun adalah simpul yang berderajat nol atau tidak memiliki anak. Pada Gambar 1, daun pohon adalah 2, 5, 11, dan 4.

Simpul Dalam

Simpul yang bukan daun dan bukan akar disebut sebagai simpul dalam. Pada Gambar 1, yang merupakan simpul dalam adalah 7, 2, 6, 5, dan 9.

3. *SUFFIX TREE*[3]

Dalam ilmu komputer (*computer science*), sebuah *suffix tree* adalah sebuah struktur data yang mempresentasikan sufiks dari string sedemikian sehingga memudahkan implementasi tertentu dengan cepat untuk berbagai operasi pada string.

Suffix tree dari string S adalah pohon yang sisi-sisinya diberi label string yang merupakan sufiks dari S dan sesuai dengan tepat satu jalur dari akar ke daun.

Dalam membangun pohon jenis ini untuk string S, akan membutuhkan waktu dan ruang linier tergantung panjang S. Jika sudah terbentuk, beberapa operasi dapat dilakukan lebih cepat, misalnya mencari substring di S, mencari substring jika berapa kesalahan kecil pada penulisan diperbolehkan, dan mencari kesamaan *regular expression*.

Suffix tree juga menyediakan salah satu solusi waktu-linear pertama untuk masalah *longest common substring*. Ini menyebabkan: menyimpan *suffix tree* sebuah string biasanya membutuhkan ruang yang jeuh lebih banyak dari pada menyimpan string itu sendiri.

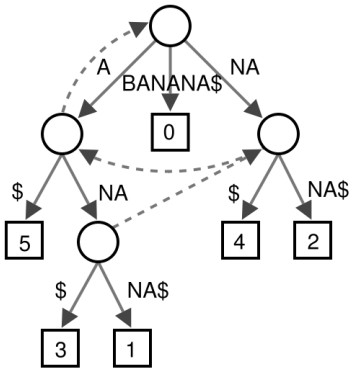
3.1 Definisi

Suffix tree untuk string S yang panjangnya *n* didefinisikan sebagai suatu pohon sedemikian sehingga:

a. Lintasan dari akar ke daun memiliki hubugan satu-satu dengan sufiks dari S.
b. Sisi-sisinya diisi string tak kosong.
c. Semua simpul (mungkin kecuali akar) memiliki minimal 2 anak.

Karena pohon ini tidak berlaku untuk semua string, maka S ditambah elemen terakhirnya dengan sebuah simbol tak terlihat, biasanya $. Hal ini akan menjamin tidak ada sufiks yang merupakan prefiks dari yang lain dan akan ada *n* daun, masing-masing untuk *n* sufiks dari S. Karena semua simpul dalam kecuali akar memiliki cabang, maka paling banyak akan ada (*n*-1) simpul, dan *n*+(*n*-1)+1=2*n* total simpul(*n* daun, (*n*-1) simpul dalam, 1 akar).

Suffix links adalah sebuah fitur kunci untuk membangun waktu-linear dari pohon. Dalam sebuah *suffix tree* yang lengkap, semua simpul dalam bukan akar memiliki sebuah *suffix links* ke simpul dalam yang lain. Jika lintasan dari akar ke suatu simpul mengeja string $\chi\alpha$, dimana χ adalah karakter tunggal dan α adalah sebuah string (mungkin kosong), maka terdapat *suffix links* ke simpul dalam yng mempresentasikan α. *Suffix tree* juga biasa digunakana dalam beberapa algoritma yang bekerja pada sebuah pohon.



Gambar 2. Contoh *Suffix Tree*

3.2 Kegunaan

Sebuah *suffix tree* unuk sebuah string S yang memiliki panjang *n* bisa dibangun dalam waktu $\Theta(n)$, jika alfabet adalah konstanta atau integer. Jika bukan, maka waktu pembuatan akan bergantung pada implementasinya.

Asumsikan bahwa sebuh *suffix tree* telah dibangun untuk string S dengan panjang *n*, atau sebuah *suffix tree* yang tergeneralisasi telah dibangun untuk sebuah set string $D = \{S_1,S_2,...,S_K\}$ dengan panjang keseluruhan $n = |n_1\backslash+|n_2|+...+|n_K|$. Kita dapat:

a. Mencari string
   - Mengecek apakah sebuah string A dengan panjang *m* adalah substring dari S dalam waktu $O(m)$.
   - Mencari kejadian awal dari susunan $A_1,...A_b$ dengan panjang keseluruhan *m* sebagai sebuah substring dalam waktu $O(m)$.
   - Mencari semua kejadian *z* dari susunan $A_1,...A_b$ dengan panjang keseluruhan *m* sebagai sebuah substring dalam waktu $O(m+z)$.
   - Menemukan sebuah *regular expression* A dalam yaktu yang diprediksikan tidak linear dalam *n.*
   - Mencari setiap sufiks dari susunan A panjang dari kesamaan terpanjang antara sebuah prefiks $A[i...m]$ dan sebuah substring di D dalam waktu $\Theta(m)$. Ini disebut sebagai *matching statistics* untuk A.

b. Mengetahui informasi string
   - Mencari *the longest common substring* dari substring S_i dan S_j dalam waktu $\Theta(n_i+n_j)$.
   - Mencari *maximal pairs*, pengulangan maksimal atau pengulangan supermaksimal dalam waktu $\Theta(m+z)$.
   - Mencari *Lempel-Ziv decompotition* dalam waktu $\Theta(n)$.
   - Mencari *the longest repeated substring* dalam waktu $\Theta(n)$.
   - Mencari kemunculan substring terbanyak dari sebuah panjang minimum dalam waktu $\Theta(n)$.
   - Mencari string terpendek dari Σ yang tidak muncul di D dalam waktu $O(n+z)$ jika ada *z* buah string.
   - Mencari string terpendek yang hanya muncul sekali dalam waktu $\Theta(n)$.
   - Mencari, untuk setiap *i*, substring terpendek dari S_i yang tidak muncul di D dalam waktu $\Theta(n)$.

Suffix tree juga dapat disiapkan untuk mendapatkan waktu konstan *lowest common ancestor* antarsimpul dalam waktu $\Theta(n)$. Kita juga bisa:

- Mencari *the longest common prefix* antara sufiks $S_i[p. .n_i]$ dan $S_j[q...n_j]$ dalam $\Theta(1)$.
- Menemukan susunan A dengan panjang *m* dengan paling banyak *k* buah ketidakcocokan penulisan dalam waktu $O(kn+z)$ dimana *z* adalah jumlah keberhasilan.
- Menemukan z palindrom maksimal dalam waktu $\Theta(n)$, atau $\Theta(gn)$ jika terdapat *gaps* sepanjang *g*, atau $\Theta(kn)$ jika terdapat *k* ketidakcocokan penulisan.
- Menemukan z *tandem repeats* dalam $O(nlog\ n+z)$ dan *k-mismatch tandem repeats* dalam $O(nlog\ (n/k)+z)$.
- Mencari *the longest common substring* untuk setidaknya *k* buah string di D dimana $k=2...K$ dalam waktu $\Theta(n)$.

3.3 Aplikasi

Suffix tree dapat digunakan untuk mencari solusi sebagian besar masalah string yang terjadi pada *text-editing*, *free-text search*, biologi perhitungan, dan bidang aplikasi lainnya. Aplikasi utama *suffix tree* diantaranya:

- Pencarian string, dengan kompleksitas $O(m)$ dimana *m* adalah panjang substring.
- Mencari *the longest repeated substring*.
- Mencari *the longest common substring*.
- Mencari palindrom terpanjang pada string.

Suffix tree biasanya digunakan dalam aplikasi *bioinformatics*, digunakan untuk mencari pola pada rangkaian DNA dan protein, yang bisa dilihat sebagai suatu string panjang. Kemampuan untuk menemukan secara efisien dengan adanya ketidak-cocokan teks mungkin adalah kekuatan terbesar *suffix tree*. *Suffix tree* juga biasa dugunakan dalam kompresi data, dimana di satu sisi digunakan untuk mencari data terulang dan di sisis lain bsa digunakan untuk tahap pengurutan pada *Burrows-Wheeler transform*. *Suffix tree* juga digunakan dalam *suffix tree clustering*, sebuah algoritma persebaran data yang digunakan pada beberapa mesin pencari.

3.4 Implementasi

Jika setiap simpul dan sisi bisa diwakilkan dalam ruang $\Theta(1)$, keseluruhan pohon dapat diwakilkan dengan ruang $\Theta(n)$. Panjang keseluruhan dari semua string pada semua sisi dalam pohon adalah $O(n^2)$, tapi setiap sisi bisa disimpan sebagai posisi dan panjang dari sebuah substring S, memberikan ruang total yang digunakan adalah $\Theta(n)$ kata. Kasus terburuk penggunaan ruang *suffix tree* terlihat sebagai sebuah string *fibbonaci*, mengisi penuh simpul sebanyak $2n$.

Pilihan penting ketika membuat sebuah implementasi *suffix tree* hubngan orang tua-anak antarsimpul. Yang biasa digunakan adalah *linked list* yang disebut *sibling list.* Masing-masing simpul memiliki penunjuk ke anak pertamanya, dan ke simpul berikutnya dalam list anak di mana anaknya berada. *Hash maps*, *sorted/unsorted array(*dengan *array doubling)*, dan *balanced search tree* biasanya juga digunakan, memberikan *running time properties* yang berbeda.

Hal yang menarik adalah

- *Cost* mencari anak jika sebuah karakter diberikan.
- *Cost* memasukan/menambah seorang anak.
- *Cost* menampilkan semua anak dari sebuah simpul.

Misalkan σ adalah ukuran alfabet, maka *cost*-nya adalah:

	Lookup	Insertion	Traversal
Sibling lists / unsorted arrays	$O(\sigma)$	$\Theta(1)$	$\Theta(1)$
Hash maps	$\Theta(1)$	$\Theta(1)$	$O(\sigma)$
Balanced search tree	$O(\log\sigma)$	$O(\log\sigma)$	$O(1)$
Sorted arrays	$O(\log\sigma)$	$O(\sigma)$	$O(1)$
Hash maps + sibling lists	$O(1)$	$O(1)$	$O(1)$

Gambar 3. Cost implementasi

Jumlah nformasi yang sangat banyak pada setiap simpul dan sisi inilah yang menyebabkan *suffix tree* sangat mahal, mengkonsumsi hampir 10-20 kali ukuran memori dari teks sumber pada implementasi yang bagus. *Suffix array* dapat mengurangi kebutuhan ini sampai faktor empat dan para penemunya telah menemukan struktur indks yang lebih kecil.

4. *SUFFIX ARRAY*[1]

Suffix array adalah pengembangan dari *suffix tree*. Dengan adanya *suffix array* ini, kita bisa mengoptimalkan fungsi-fungsi yang ada pada *suffix tree* yang terhalang oleh keterbatasan kompleksitas.

Suffix array adalah struktur data yang simpel yang memudahkan pencarian substring pada teks dan memudahkan pengidentifikasian teks yang berulang.

Sebuah *suffix array* lebih kompak daripada sebuah *suffix tree* dan mengijinkan penyimpanan pada *secondary memory*.

4.1 Definisi

Suffix array adalah truktur data yang didisain untuk mengoptimalkan dan mengefisienkan pencarian pada teks besar. Struktur datanya sesimpel suatu larik/array yang mengandung semua penunjuk ke sufiks teks yang diurutkan secara *lexicographic* atau secara alfabetik. Masing-masing sufiks adalah string yang dimulai pada posisi tertentu pada teks dan berakhir di akhir teks.

Melakukan pencarian pada sebuah teks dapat dilakukan dengan *binary search* menggunakan *suffix array*.

4.2 Membangun *Suffix Array*

Misalkan kita meiliki teks sampel “abracadabra” dan akan membangun *suffix array* darinya.

Pertama, kita tetapkan indeks point ke teks sampel. Indeks point menentukan posisi sehingga pencarian dapat dilakukan. Dalam hal ini, indeks dipaangkan pada karakter per karakter. Setelahnya, kita bisa melakukan pencarian pada teks tersebut dengan suffix array dengan posisi awal di mana saja sesuai kebutuhan kita.

Text	a	b	r	a	c	a	d	a	b	r	a
Index	0	1	2	3	4	5	6	7	8	9	10

Gambar 4. Pemasangan indeks

Kedua, urutkan indeks point berdasar kepada masing-masing sufiks yang sesuai. Kesesuaian antara indeks dan sufiks dapat diamati pada gambar di bawah ini.

Suffix											Index
a	b	r	a	c	a	d	a	b	r	a	0
	b	r	a	c	a	d	a	b	r	a	1
		r	a	c	a	d	a	b	r	a	2
			a	c	a	d	a	b	r	a	3
				c	a	d	a	b	r	a	4
					a	d	a	b	r	a	5
						d	a	b	r	a	6
							a	b	r	a	7
								b	r	a	8
									r	a	9
										a	10

Gambar 5. Pemasangan indeks pada sufiks

Sorted Suffix											Index
a											10
a	b	r	a								7
a	b	r	a	c	a	d	a	b	r	a	0
a	c	a	d	a	b	r	a				3
a	d	a	b	r	a						5
b	r	a									8
b	r	a	c	a	d	a	b	r	a		1
c	a	d	a	b	r	a					4
d	a	b	r	a							6
r	a										9
r	a	c	a	d	a	b	r	a			2

Gambar 6. Indeks setelah sufiks diurutkan

Akhirnya, indeks hasil pengurutan sufiks menjadi *suffix array* untuk teks sampel “abracadabra”.

Biasanya suatu teks akan diakatakan berakhir jika ada suatu penanda. Dalam al ini, penanda tersebut berupa simbol. Sama seperti *suffix tree*, simbol yang biasanya digunakan adalah $. Penambhan simbol ini akan berpengaruh pada panjang teks. Panjang teks akan bertambah satu dari panjang semula, namun dengan adanya penanda ini, beberapa proses akan menjadi lebih mudah.

Dengan menyertakan simbol $ sebagai akhiran, teks sampel akan menjadi “abracadabra$”. Sehingga sufiks

yang telah diurutkan berdasarkan abjad adalah seperti di bawah ini.

11	$
10	a$
7	abra$
0	abracadabra$
3	acadabra$
5	adabra$
8	bra$
1	bracadabra$
4	cadabra$
6	dabra$
9	ra$
2	racadabra$

Dengan demikian, suffix array untuk contoh teks sampel tersebut diatas adalah { 11,10,7,0,3,5,8,1,4,6,9,2 }.

4.3 Algoritma

Cara paling mudah dalam membangun *suffix array* adalah dengan menggunakan sebuah algoritma *comparison sort* yang efektif. Hal ini membutuhkan O(*n*log*n*) perbandingan sufiks, tetapi perbandingan sufiks membutuhkan waktu O(*n*). Sehingga keseluruhan *runtime* dari pendekatan ini adalah O(n^2log*n*). Algoritma yang lebih canggih akan dapat mengembangkan kemampatan menjadi O(*n*log*n*) dengan mengeksploitasi hasil *partial sort* untuk menghindari perbandingan yang berlebihan. Beberapa algoritma Θ(*n*) juga sudah dikembangkan sehingga mampu melakukan pembangunan *suffix array* lebih cepat dan membutuhkan jumlah ruang yang digunakan O(*n*) dengan nilai konstanta ang kecil.

Pengembangan oleh Salson *et al.* mengusulkan sebuah algoritma untuk memperbaharui (*updates*) *suffix array* dari sebuah teks yang telah diedit daripada membangun ulang sebuah *suffix array* yang baru dari awal. Walaupun secara teori, kasus terburuk kompleksitas waktu adalah O(*n*log*n*), dapat diamaati bahwa *suffix array* ini berjalan jauh lebih baik daripada metode tercepat yang lain.

4.4 Aplikasi

Suffix array dari sebuah string dapat digunakan sebagai sebuah indeks untuk mencari secara cepat setiap peristiwa dari sebuah substring di dalam sebuah string. Mencari setiap peristiwa dari sbuah substring ekivalen dengan mencari setiap sufiks yang dimulai dengan substring tersebut. Berkat pengurutan secara alfabet, sufiks-sufiks ini akan dikumpulkan dalam *suffix array*, dan bisa diketemukan secara efisien dengan *binary search*.

Jika diimplementasikan secara lugas, pencarian dengan menggunakan *binary search* ini membutuhkan waktu O(*m*log*n*), dimana *m* adalah panjang substring. Untuk mencegah perbandingan berulang, struktur data ekstra memberikan informasi tentang *the longest common prefixes*(LCPs) dari sufiks-sufiks yang dibentuk, memberikan waktu pencarian O(*m*+log*n*).

Algoritma pengurutan sufiks dapat digunakan untuk melakukan *the Burrows-Wheeler transform* (BWT). Secara teknik, BWT membutuhkan pengurutan *cylic permutation* dari sebuah string, bukan sufiks. Kita dapat memperbaikinya dengan cara menambahkan ke dalam string karakter penanda suatu akhir string yang diurutkan secara *lexicographical* sebelum setiap katakter yang lain diurutkan. Mengurutkan *cyclic permutation* ekivalen dengan mengurutka sufiks.

Contoh pengurutan *cyclic permutation* tersebut yaitu sebagai berikut.

11	$abracadabra
10	a$abracadabr
7	abra$abracad
0	abracadabra$
3	acadabra$abr
5	adabra$abrac
8	bra$abracada
1	bracadabra$a
4	cadabra$abra
6	dabra$abraca
9	ra$abracadab
2	racadabra$ab

Dari contoh di atas, kolom terakhir dari rangkaian huruf-huruf tersebut adalah “ard$rcaaaabb”. Inilah yang disebut dengan *the Burrows-Wheeler transform*.

IV. KESIMPULAN

Pengolahan string dengan berbagai operasi sangat dibutuhkan untuk berbagai keperluan. Salah satu cara mudah mengolah string adalah dengan menggunakan *suffix tree*. Namun ternyata, kompleksitas *suffix tree* masih terlalu rumit. *Suffix array* hadir sebagai salah satu solusi untuk mengoptimalkan fungsi yang dapat dilakukan oleh *suffix tree*.

REFERENSI

[1] Udi Manber & Gene Myers, “Suffix arrays: a new method for on-line string searches”, SIAM Journal on Computing, Volume 22, Issue 5, October 1993, pp. 935-948.

[2] Rinaldi Munir, “Diktat Kuliah IF2091, Struktur Diskrit”, Program Studi Teknik Informatika, STEI ITB, 2008.

[3] Wikipedia, Wikipedia – Free Encyclopedia, www.wikipedia.com, 2009. Tanggal akses: 18 Desember 2009, pukul 20.00 WIB.