Suffix Array dan Kegunaannya Dalam Memecahkan Berbagai Persoalan String

Christianto - NIM : 1350003[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13510003@std.stei.itb.ac.id, handojo_christianto@yahoo.com

Abstract*—Makalah ini akan membahas salah satu metode yang banyak dipakai untuk memecahkan berbagai macam persoalan string, yaitu suffix array. Pembahasan mencakup konsep dasar dari suffix array, algoritma-algoritma pembuatan suffix array, algoritma untuk pembuatan longest common prefix dari suffix array, serta contoh-contoh persoalan string yang dapat diselesaikan dengan penggunaan suffix array.

Index Terms*—suffix array, string, longest common prefix.

1. Pendahuluan

Problem *string matching* adalah salah satu problem yang memiliki cukup banyak kegunaan dalam kehidupan sehari-hari serta dalam bidang penelitian biologi. Banyak algoritma yang sudah dikembangkan untuk permasalahan ini, seperti Knuth-Morris-Prath, Boyer-Moore, Aho-Corasick, dll. Hanya saja, algoritma-algoritma ini memiliki kelemahan berupa sulitnya mengembangkan algoritma-algoritma ini untuk memecahkan tipe-tipe permasalahan lain yang juga menyangkut string, seperti power string, smallest rotating string, dll. Kelemahan lainnya adalah kebanyakan algoritma di atas melakukan preprocessing pada pattern yang dicari, bukan pada text, sehingga preprocessing harus dilakukan berkali-kali bila ada banyak pencarian.

. Makalah ini akan membahas mengenai sebuah struktur data yang dapat menyelesaikan berbagai macam persoalan string, yaitu *suffix array*. Makalah ini juga akan membahas cara pembuatan *suffix array* yang efisien serta contoh penggunaannya untuk pemecahan permasalahan.

2. Penjelasan Mengenai *Suffix Array*

Konsep dari *suffix array* diperkenalkan oleh Manber dan Myers pada paper yang diterbitkan tahun 1990[1] sebagai alternatif dari struktur data lain yang serupa, tetapi memakan jauh lebih banyak memori, yaitu *suffix tree*. Pada dasarnya, *suffix array* adalah sebuah array dengan banyak elemen sesuai panjang string, yang berisi sufiks-sufiks dari string yang diberikan setelah dilakukan pengurutan secara leksikografis. Untuk menghemat memori, sufiks-sufiks ini dilambangkan dengan indeks tempat sufiks dimulai. Sebagai contoh, *suffix array* dari string "algoritma" adalah {9,1,3,6,2,8,4,5,7}, yang apabila dilihat sebagai sufiks berisi {"a","algoritma","goritma", "itma","lgoritma","ma","oritma","ritma","tma"}.

Konsep yang sederhana ini ternyata dapat digunakan untuk memecahkan berbagai macam persoalan string dengan mangkus apabila digabungkan dengan sebuah struktur data lain, yaitu *Longest Common Prefix*(LCP) *Array*. LCP Array sendiri menyimpan nilai prefiks terpanjang yang bisa didapat dari sebuah sufiks dalam *suffix array* dengan sufiks pada indeks sebelumnya. Pada contoh di atas, isi dari LCP Array adalah {0,1,0,0,0,0,0,0,0}.

3. Algoritma-algoritma Pembuatan *Suffix Array*

3.1. Algoritma O(N^2 lg N)

Cara pembuatan yang paling mudah adalah membuat semua sufiks yang ada dari string yang diberikan, kemudian melakukan pengurutan dengan quicksort atau sejenisnya untuk mendapatkan sufiks yang sudah terurut. Setelah itu dilakukan pencarian indeks yang sesuai dengan sufiks itu untuk menghasilkan *suffix array* yang diinginkan. Kompleksitas waktu dari algoritma ini sekilas adalah O(N lg N) dari waktu yang diperlukan untuk pengurutan, tetapi yang benar adalah O(N^2 lg N), karena setiap operasi perbandingan sufiks memiliki kompleksitas O(N). Sebagai contohnya adalah string "aaaaaaaaaaaaa".

3.2. Algoritma O(N (lg N)^2)

Cara yang lebih baik untuk membuat *suffix array* pada dasarnya sama dengan cara di atas, tapi dilakukan sedemikian sehingga operasi perbandingan dapat dilakukan secara konstan. Caranya adalah mengurutkan awalan-awalan sufiks terlebih dahulu dengan panjang 2^i, kemudian memakai hasil pengurutan itu untuk mempercepat pengurutan awalan sufiks dengan panjang 2^{i+1}. Pengurutan selesai setelah panjang awalan sufiks yang dipakai minimal sama dengan panjang string. Kasus basisnya adalah panjang awalan sufiks 1, yaitu karakter itu sendiri.

Proses pencepatan perbandingan antara 2 buah awalan sufiks dilakukan dengan membandingkan hasil pengurutan

dari paruh awal awalan sufiks, baru kemudian dengan paruh akhir awalan sufiks. Karena hasil pengurutan ini berupa integer, maka perbandingan ini memiliki kompleksitas waktu konstan.

Sebagai contoh, proses pembuatan *suffix array* dari kata "banana" adalah :

- Step 1 : 2 1 3 1 3 1, karena karakter yang ada adalah {a,b,n}.
- Step 2 : 3 2 4 2 4 1, yaitu untuk {ba, an, na, an, na, a}.
- Step 3 : 4 3 6 2 5 1, yaitu untuk {bana, anan, nana, ana, na, a}
- Step 4 : 4 3 6 2 5 1, yaitu untuk {banana, anana, nana, ana, na, a}

Setelah proses pengurutan di atas selesai, sufiks array dapat dibuat dengan cepat, karena isi setiap indeks array adalah lokasi sufiks tersebut dalam *suffix array*. Hasil akhirnya adalah {6, 4, 2, 1, 5, 3}.

Cara perbandingan di atas memiliki kompleksitas waktu sebesar $O(N (\lg N)^2)$, karena terdapat lg N tahap pengurutan, dimana setiap pengurutan memiliki kompleksitas waktu O(N lg N).

3.3. Algoritma O(N lg N)

Algoritma di atas dapat diperbaiki dengan pengamatan bahwa urutan hasil pengurutan sebelumnya tidak akan memakai angka lebih besar dari N, yaitu panjang string yang sedang dibuat *suffix array*-nya. Karena itu teknik pengurutan yang tidak berbasis pada perbandingan dapat dipakai, seperti *counting sort* atau *radix sort*. Dengan cara ini, kompleksitas waktu pengurutan tiap tahap adalah O(N), sehingga kompleksitas waktu secara keseluruhan adalah O(N lg N). Berikut ini adalah pseudocode implementasi dengan memakai struktur data linked-list.

```
Var belakang, depan : list[1..panjang]

tahap ← 1
while (tahap < panjang) do
    Clear belakang and depan
    for I ← 1 to panjang do
        Append I to belakang[pos[i+tahap]]

    For I ← 1 to panjang do
        Foreach (j in belakang[i]) do
            Append j to depan[pos[j]]

    temu ← 0
    For I ← 1 to panjang do
        Foreach (j in depan[i]) do
            pos[j] ← temu
            if (data(j) != data(prev)) then
                temu ← temu+1
            prev ← j

    tahap← tahap * 2
```

Pseudocode yang diberikan di atas masih sangat kasar, oleh karena itu penulis menyarankan paper [2] sebagai acuan bagi mereka yang ingin mencoba untuk membuat program sendiri.

3.4. Algoritma O(N)

Algoritma pembuatan suffix array dengan kompleksitas waktu linear sudah ada, salah satunya adalah membuat *suffix tree* terlebih dahulu dengan algoritma yang linear, seperti Algoritma Ukkonen, kemudian dilakukan traversal dengan depth-first search pada tree tersebut. Hanya saja cara ini sebetulnya sangat tidak disarankan, karena *suffix tree* sendiri memiliki kemampuan yang setara dengan *suffix array* yang sudah ditambah LCP Array.

Banyak paper yang membahas algoritma pembuatan *suffix array* dengan kompleksitas waktu linear. Salah satunya adalah paper dari Karkkainen dan Sanders(2003) [3]. Cara pembuatannya berdasarkan pada prinsip divide and conquer. Karena inti algoritma ini cukup sulit untuk dibahas di makalah ini, penulis menyarankan untuk membaca [3] sebagai acuan.

3.5. Algoritma Pembuatan LCP Array

Untuk pembuatan LCP Array dalam waktu linear, berikut ini adalah contoh kode dalam C++. Kode ini memerlukan data lokasi dalam *suffix array* untuk sufiks yang dimulai dari posisi i. Data lokasi ini sudah terhitung langsung dengan pseudocode yang diberikan di atas.

```
LCP-array(N)

  height[0] = 0;
  for (int i=0,h = 0;i<panjang;++i) {
    if (pos[i] > 0) {
      int j = urut[pos[i]-1];
      while ((i+h < panjang)&&(j+h <
panjang)&&(kata[i+h]==kata[j+h])) ++h;
      height[pos[i]] = h;
      if (h) --h;
    }
  }

/*pos menyimpan lokasi sufiks ke-i dalam suffix
array. Height menyimpan LCP Array. Urut adalah
suffix array.*/
```

Kompleksitas waktu dari algoritma di atas adalah linear, untuk pembuktian lebih lanjut pembaca disarankan untuk mengacu [4], karena pembuktian yang diberikan cukup rumit dan panjang sehingga tidak bisa dimuat dalam makalah ini.

4. Persoalan yang Dapat Diselesaikan

Berikut ini akan diberikan daftar-daftar persoalan yang dapat diselesaikan secara mangkus dengan menggunakan *suffix array* dan LCP array. Penulis juga akan menuliskan cara-cara lain yang diketahui untuk menyelesaikan persoalan ini.

4.1. *String Searching*

Kegunaan pertama suffix array adalah memproses text yang diberikan sehingga dapat mencari pattern yang diberikan dalam waktu linear, tanpa perlu melakukan preprocessing pada pattern yang diberikan. Berikut ini cara pencarian yang memiliki kompleksitas waktu linear sesuai dengan panjang pattern dan panjang text :

1. Hitung LCP antara suffix array indeks 1 dan pattern yang diberikan. Simpan dalam variable bernama cocok.
2. Untuk setiap i dalam suffix array dari 2 hingga panjang text, lakukan proses di bawah ini:
   - Bila cocok < height[i], pertahankan nilai cocok.
   - Bila cocok > height[i], ubah nilai cocok menjadi nilai height[i].
   - Bila cocok = height[i], lakukan looping yang berfungsi menambah nilai cocok selama karakter ke-cocok+1 pada pattern dan sufiks yang bersangkutan pas.
3. Untuk setiap indeks dimana cocok memiliki nilai sama dengan panjang pattern, berarti disitu ditemukan pattern.

Cara di atas sebetulnya mensimulasikan proses pencarian pattern pada *suffix tree*, yang jauh lebih mudah karena hanya perlu melakukan traversal node-node pada tree.

Tentu saja, cara ini hanya berlaku apabila text yang diberikan tidak berubah selama pemrosesan, karena perubahan text bisa mengubah *suffix array* secara drastis, dan pembahasan cara melakukan update dengan cepat berada di luar batasan makalah ini.

Untuk algoritma dengan kompleksitas waktu dibawah O(M+N), pembaca bisa mengacu pada [1], dimana diberikan algoritma string searching dengan kompleksitas waktu O(M + log N). Untuk hasil yang lebih baik lagi, gunakan *suffix tree* untuk mendapatkan algoritma dengan kompleksitas waktu O(M), terbaik yang bisa diharapkan.

Cara ini lebih superior dari KMP, Boyer-moore dan algoritma lainnya karena preprocessing cukup dilakukan sekali saja, yaitu pada text.

4.2. Longest Common Substring

Persoalan Longest Common Substring adalah mencari substring terpanjang yang muncul pada 2 buah string yang diberikan. Persoalan ini dapat digeneralisasi menjadi mencari Longest Common Substring dari K buah string yang diberikan, dimana panjang masing-masing string bisa tidak sama.

Salah satu cara menyelesaikan persoalan ini adalah mencoba semua substring yang mungkin dari string pertama dalam masukan, kemudian menggunakan algoritma string searching yang linear untuk mengecek kemunculan substring dalam string tersebut. Cara ini memerlukan waktu $O(K*max(N)^3)$, dimana max(N) berarti adalah panjang string yang paling besar dalam masukan.

Pengembangan lebih lanjut dari cara di atas dapat dilakukan apabila algoritma string searching yang dipilih adalah KMP, karena preprocessing pattern dapat dilakukan secara bertahap, sesuai dengan substring yang sedang dibuat. Pencarian substring dalam setiap string juga dapat dilakukan dengan kompleksitas waktu agregat O(N), sehingga kompleksitas waktunya adalah $O(K*max(N)^2)$.

Cara yang akan penulis berikan berbasis pada *suffix array*. Idenya adalah menggabungkan semua string menjadi sebuah string saja, dimana titik perpisahan antar string ditandai dengan sebuah karakter yang tidak muncul dalam alfabet string. Contohnya untuk string masukan apel, jeruk, melon, pisang dan semangka, hasil gabungannya adalah apel$jeruk$melon$pisang$semangka. Fungsi dari penggunaan karakter khusus ini adalah memastikan isi dari LCP array tidak akan berisi prefix yang memintas batasan antar string.

Setelah penggabungan string selesai dilakukan, tahap berikutnya adalah membangun *suffix array* dan LCP array dari string gabungan itu. Untuk tahap berikutnya, gunakan pseudocode ini :

```
Longest-Common-Substring(N,K)

isi ← 1
Indeks ← 1
Jawab ← 0
for i ← 1 to length(N) do
    while (isi < K and indeks <= length(N)) do
        if (aktif[asal[i]] = 0) then
            inc(aktif[asal[indeks]]) ;
            isi ← isi + 1
        indeks ← indeks + 1

    if (isi < K) then
        Break;

    jawab ← max(jawab, minimum of LCP between
                        i+1 and indeks-1)
    dec(aktif[asal[i]])
    if (aktif[asal[i]] = 0) then
        isi ← isi - 1

→ jawab
```

Algoritma diatas akan memberikan Longest Common Prefix untuk semua string pada masukan. Permasalahan

yang muncul adalah bagaimana cara mencari minimum dari LCP untuk rentang i+1 hingga indeks-1 untuk setiap i. Solusi untuk permasalahan ini dapat menggunakan struktur data tambahan seperti segment tree atau range minimum query yang dibuat dengan dynamic programming. Hanya saja hasilnya adalah algoritma yang memiliki kompleksitas waktu O(N lg N), belum lagi kesulitan membuat segment tree atau range minimum query memungkinkan terjadinya kesalahan dalam pembuatan program.

Cara yang berikut ini memiliki kompleksitas waktu linear, sehingga akan didapat algoritma yang memiliki kompleksitas waktu linear secara keseluruhan. Cara ini memanfaatkan sebuah struktur data yang disebut double-ended queue, atau bisa disingkat deque. Struktur data ini adalah queue yang bisa diproses dari kedua ujung. Triknya adalah mempertahankan isi deque tetap menaik, bahkan ketika elemen-elemen baru dimasukkan dari salah satu ujung deque, sebut saja ekor.

Pada algoritma di atas, pada loop while ketika dilakukan penambahan indeks, nilai LCP[indeks] juga dimasukkan ke dalam deque melalui ekor. Pada proses pemasukan nilai baru ini, semua nilai lama pada ekor yang bernilai lebih besar dari LCP[indeks] dibuang, sehingga hasil akhirnya properti nilai deque tetap terjaga. Kemudian nilai pada kepala deque dibuang apabila elemen itu berasal dari LCP[i]. Sekarang nilai minimum LCP akan bisa ditemukan pada kepala deque. Karena setiap elemen maksimal dimasukkan sekali dan dikeluarkan sekali dari deque, kompleksitas waktunya adalah linear.

Solusi untuk permasalahan ini memang tidak murni memakai *suffix array*, tetapi dengan cara ini didapat algoritma yang kompleksitas waktunya linear dengan panjang semua string.

4.3. Membangun *Suffix Tree*

Kegunaan ini seharusnya tergolong wajar, karena kedua struktur data ini memang serupa, sehingga dari satu struktur data, kita bisa membangun struktur data yang lain. Kuncinya adalah pemanfaatan LCP array sehingga bisa dilakukan traversal karakter-karakter pada *suffix array* secara efisien selama proses pembangunan *suffix tree*. Permasalahan yang ada tinggal memilih struktur data mana yang akan dipakai.

4.4. Power String

Definisi power string adalah string yang terbentuk dari sebuah string lain yang diulang sebanyak K kali. Contohnya adalah kalikalikali merupakan power string, karena kalikalikali dibentuk dari kali yang diulang sebanyak 3 kali. Permasalahan yang diberikan adalah menentukan nilai K terbesar yang mungkin untuk sebuah string sedemikian sehingga string asal ini adalah hasil pengulangan sebuah string sebanyak K kali. K mungkin saja 1.

Solusi dengan menggunakan suffix array adalah membangun *suffix array* serta LCP array untuk string masukan, kemudian dari hasilnya cari posisi string awal pada *suffix array*. Dari indeks itu lakukan looping hingga indeks 1 untuk mencari nilai terkecil pada suffix[i] sedemikian sehingga selisih antara panjang suffix[i] dan panjang string masukan dapat membagi panjang string masukan. Nilai ini adalah panjang string yang diulang, untuk mencari K kita dapat membagi panjang string dengan nilai yang didapat ini.

Solusi lain yang mungkin adalah menggunakan border function dari KMP untuk mencari daftar nilai-nilai yang mungkin menjadi panjang string yang diulang. Kedua solusi ini memiliki kompleksitas waktu linear.

4.5. Smallest Rotating String

Diberikan sebuah string, kita dapat memutar string tersebut sesuka hati, dimana operasi memutar didefinisikan sebagai mengambil karakter pertama string dan menaruhnya di belakang string. Sebagai contoh, untuk string "abc", string yang bisa terbentuk dengan melakukan operasi memutar adalah "abc", "bca", dan "cab". Kita diminta mencari string yang terkecil secara leksikografis.

Cara yang efektik adalah menggabungkan string tersebut dengan dirinya sendiri, kemudian membangun *suffix array* untuk string hasil penggabungan tersebut. Setelah itu lakukan pencarian indeks pertama suffix array yang isinya bernilai antara 1 hingga N, dimana N adalah panjang string. Sufiks ini berisi string terkecil yang kita cari.

Solusi lain yang mungkin adalah menggunakan Algoritma Booth, yang merupakan modifikasi KMP. Kompleksitas waktu keduanya adalah O(N), tetapi algoritma Booth tidaklah intuitif, sehingga lebih susah diingat dan diprogram.

4.6. Problem-Problem Lain

Daftar yang diberikan disini tidaklah lengkap. Masih banyak persoalan string lainnya yang dapat dipecahkan dengan menggunakan *suffix array* maupun *suffix tree*. Bagi para pembaca yang berminat untuk mencari problem-problem yang dapat dipecahkan dengan kedua struktur data ini, dapat mencoba [2] maupun [5], walaupun [5] tidak memberikan solusi secara langsung. [5] juga dapat dijadikan acuan untuk mempelajari algoritma-algoritma yang berhubungan dengan string.

5. Kesimpulan

Suffix array adalah struktur data yang dikembangkan untuk mengatasi permasalahan pemakaian memori yang sangat besar yang terjadi pada suffix tree. Karena keduanya memiliki basis yang sama, maka kedua struktur data ini dapat dikonversi satu sama lain sesuai keperluan.

Ada banyak tipe-tipe permasalahan yang dapat dipecahkan dengan memakai suffix array maupun suffix

tree. Keunggulannya adalah hampir tidak ada perubahan kode untuk pembuatan suffix array dari permasalahan-permasalahan yang dibahas di atas, penambahan kode hanya dilakukan untuk melengkapi data-data yang diperlukan untuk menyelesaikan permasalahan. Dengan tingkat adaptasi yang luar biasa besar ini, struktur data ini layak untuk dipelajari lebih lanjut.

6. Referensi

[1] Manber, Udi; Myers, Gene (1990). "Suffix arrays: a new method for on-line string searches." *In Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, 90(319): 327.

[2] Vladu, Adrian; Negruseri, Cosmin. "Suffix Arrays - A Programming Contest Approach". GInfo 15/7. November 2005.

[3] Karkkainen, Juha; Sanders, Peter. "Simple Linear Work Suffix Array Construction". ICALP 2003, LNCS 2719, page 943-955. 2003.

[4] Kasai, T.; Lee, G.; Arimura, H.; Arikawa, S.; Park, K. (2001). "Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications". *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*. Lecture Notes in Computer Science. 2089. pp. 181-192.

[5] Gusfield, Dan (1999). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. USA: Cambridge University Press.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Desember 2012

Christianto - 13510003