Optimasi Searching Kata Menggunakan Pohon

Fikri Aulia (13513050)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13513050@std.stei.itb.ac.id

***Abstrak*—*Searching*/Pencarian merupakan sebuah permasalahan yang umum terjadi terutama dalam dunia computer. Terdapat banyak algoritma untuk menyelesaikan permasalahan ini dan tentunya dengan kompleksitas yang berbeda-beda. Tetapi sebuah algoritma yang memiliki kompleksitas yang lebih kecil belum tentu lebih cepat dibandingkan dengan algoritma dengan kompleksitas yang lebih besar. Hal ini dapat terjadi karena kondisi data yang dicari lebih menguntungkan bagi algoritma yang memiliki kompleksitas yang lebih besar tersebut. Sehingga penggunaan algoritma yang tepat terhadap suatu data juga mempengaruhi kecepatan pencarian data.**

***Kata Kunci*—Searching, kompleksitas, *Binary Search, Merge Search*.**

I. PENDAHULUAN

Pencarian data merupakan sebuah permasalahan yang umum terjadi dalam dunia informatika sehingga berbagai algoritma ditemukan dengan harapan permasalahan perncarian dapat diselesaikan dengan lebih efesian dan cepat. Salahsatu cara pengelesaian masalah pencarian adalah dengan menggunakan Pohon. Contoh algoritma yang ada dengan menggunakan pohon adalah *Binary Search* dan *Merge Search*.

Namun dalam beberapa permasalahan, algoritma pencarian yang yang secara umum lebih cepat belum tentu lebih cepat dibandingkan suatu algoritma tertentu yang secara umum lebih lambat. Hal ini memungkinkan terjadi karena data yang diolah memiliki kondisi yang lebih baik bagi algoritma tersebut. Oleh karena itu, modifikasi dari suatu algoritma untuk menyesuaikan dengan data yang diolah akan sangat membantu dalam meningkatkan kecepatan pencarian suatu data.

Salah satu permasalahan yang berhubungan dengan pencarian data adalah menentukan apakah suatu kata terdapat dalam suatu kamus kata atau tidak. Permasalahan ini membutuhkan algoritma yang tepat mengingat kira diharuskan menentukan ada tidaknya suatu kata diantara ratusan ribu kosakata. Jika terdapat kesalahan dalam menentukan algoritma yang dipakai. Maka akan dapat menyebabkan lamanya proses pencarian.

II. TEORI

2.1. Definisi Pencarian

Pencarian adalah suatu proses untuk menemukan data dari sekumpulan data yang ada. Data yang dicari dapat berupa bilangan, kata, dan lain-lain.

2.2 Macam-Macam Algoritma Pencarian

2.2.1 Linier Search

Linier Search adalah suatu algoritma pencarian yang mengecek semua data yang ada secara sekuensial. Linier Search merupakan algoritma pencarian yang paling sederhana dan memiliki kompleksitas O(n)

2.2.2 Binary Search

Binary Search adalah algoritma dengan menggunakan Pohon yang memiliki dua anak. Dimana anak kiri memiliki nilai yang lebih kecil daripada anak yang berapa disebelah kanan. Metode pencarian menggunakan binary search memiliki kompleksitas O(log n).

Algoritma binary search bekerja hanya pada data yang sudah terurut. Sehingga untuk mendapatkan penyesaian terbaik juga membutuhkan algoritma pengurutan yang baik pula jika data yang diolah belum terurut.

Implementasi dari algoritma binary search adalah sebagai berikut

```
boolean binary_tree(tree T, infotype X){
        switch (T) {
                Leaf(T) : {return false;}
                (T == X): {return true;}
                (T > X)  : {return
binary_tree(Left(X), X);}
                (T < X)  : {return
binary_tree(Right(X), X);}
        }
}
```

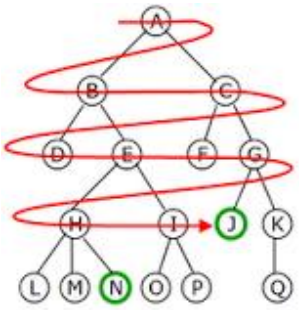
2.2.3 Breadth First Search

Algoritma Breadth First Search adalah sebuah algoritma yang dilakukan dengan mengunjungi semua node pada level n dari kiri ke kanan. Baru setelah itu mengunjungi node yang berada pada posisi n+1 dengan cara yang

sama yaitu dari kiri ke kanan. Kemudian pindah ke level berikutnya sampai data yang dicari ditemukan.

Adapun kompleksitas dari algoritma ini bergantung pada kondisi data yang akan dicari. Sehingga pengondisian data berpengaruh terhadap algoritma ini.

Berikut gambaran bagaimana Breadth First Search bekerja :



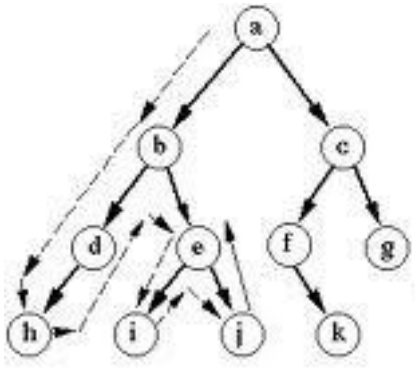
http://4.bp.blogspot.com/_NlIoEapkArE/Sy8_sUEtdhI/AAAAAAAABew/CxdGc03jLZE/s320/BFS.png
11/10/2014

2.2.4 Depth First Search

Algoritma Depth First Search adalah sebuah algoritma yang melakukan pencarian ke salah satu sub tree, kemudian dilanjutkan ke subtree dari subtree tersebut sampai ke daun terakhir. Ketika sudah sampai ke daun terakhir, maka akan dilanjutkan pengecekan pada daun selanjutnya.

Sama seperti algoritma Breadth First Search, kompleksitas dari algoritma ini bergantung pada kondisi data yang akan dicari. Sehingga pengondisian data berpengaruh terhadap algoritma ini.

Berikut bagaimana Depth First Search bekerja :



Depth-first search

http://lh4.ggpht.com/-fpq8cuVmU5o/T_2HQS5ZfxI/AAAAAAAAAcI/-fzXGpfcwfQ/depth%252520-%252520search_thumb%25255B5%25255D.jpg?imgmax=800
11/10/2014

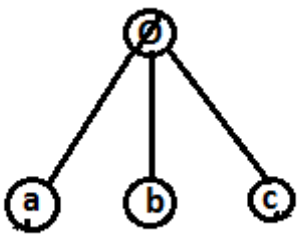
III. MENENTUKAN ALGORITMA YANG TEPAT UNTUK PENCARIAN KATA

. 3.1 Penyimpanan Data Kamus

Dikarenakan kondisi dari data yang akan dicari menentukan kompleksitas dari algoritma pencarian yang kita gunakan. Maka, pengondisian data yang akan dicari menjadi krusial.

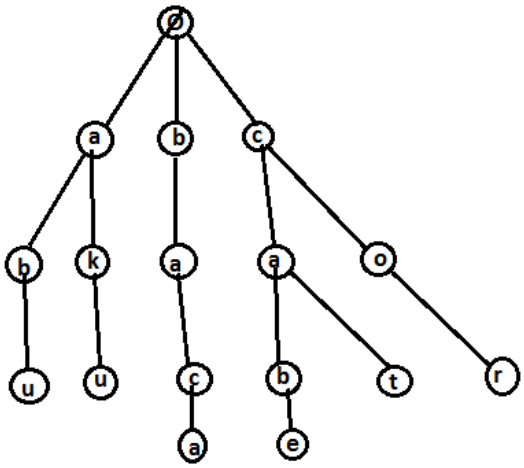
Data yang akan diolah terlebih dahulu dimasukkan kedalam sebuah list linier dengan tipe char. Dengan node awal(level 0) dari list linier kamus tersebut adalah kosong. Kemudian node pada level 0 menunjuk ke semua data yang ada pada level 1. Data pada level 0 adalah semua kemungkinan huruf awal dari semua kata yang ada di dalam kamus.

Berikut ilustrasi pohon pada level 0 dan 1 jika isi kamus adalah ‘aku’, ‘abu’, ‘baca’, ‘cabe’, ‘cat’, dan ‘cor’:



Setelah itu, pada list level 1 menunjuk ke semua kemungkinan huruf kedua dari huruf pertama. Misalkan huruf pertama adalah ‘a’, maka semua kemungkinan huruf kedua adalah ‘k’ dan ‘b’. sehingga anak dari node ‘a’ adalah ‘k’ dan ‘b’. dan proses yang sama dilakukan pada

level-level selanjutnya sehingga dibentuk data dengan ilustrasi pohon sebagai berikut :



Adapun bentuk tipe bentukan penyimpan Tree tersebut salahsatunya dapat dibuat seperti berikut :

```
typedef struct tHash *HashAddress;
typedef struct tHash{
        HashAddress NextA;
        HashAddress NextB;
        HashAddress NextC;
        HashAddress NextD;
        HashAddress NextE;
        HashAddress NextF;
        HashAddress NextG;
        HashAddress NextH;
        HashAddress NextI;
        HashAddress NextJ;
        HashAddress NextK;
        HashAddress NextL;
        HashAddress NextM;
        HashAddress NextN;
        HashAddress NextO;
        HashAddress NextP;
        HashAddress NextQ;
        HashAddress NextR;
        HashAddress NextS;
        HashAddress NextT;
        HashAddress NextU;
        HashAddress NextV;
        HashAddress NextW;
        HashAddress NextX;
        HashAddress NextY;
        HashAddress NextZ;
}HashTree;
typedef HashAddress Hash;
```

3.2 Pencarian Data

Setelah terbentuk data dengan kondisi yang telah dibahas diatas. Maka dilanjutkan dengan proses pencarian.

Proses pencarian data menggunakan prinsip Depth First Search. Namun dengan sedikit perubahan.

Proses pencarian dilakukan dengan membandingkan huruf pertama dari kata yang akan dicari dengan anak dari node pada level 0. Pada proses ini dibutuhkan sebuah algoritma tertentu untuk mengecek apakah huruf pertama dari kata yang dicari merupakan anak dari node 0.

Jika ditemukan anak dari node di level 0 pada level 1. Dan kata yang dimaksud bukanlah kata yang dicari, maka pencarian akan dilakukan dengan huruf pada urutan berikutnya pada subtree dari anak huruf pertama dan dilakuan proses yang sama sampai kata yang akan dicari ditemukan.

Berikut implementasi dari algoritma pencarian dengan menggunakan C :

```
boolean IsMemberKamus (char word[20],
HashAddress FirstHash, int nchar){
   HashAddress P = FirstHash;
   if (P == Nil) {
      if ((nchar+1) == strlen(word)){
         return true;
      } else {
         return false;
      }
    } else {
       switch (word[nchar]){
            case 'A' : {return IsMemberKamus(word,
P->NextA, nchar+1);}
            case 'B' : {return IsMemberKamus(word,
P->NextB, nchar+1);}
            case 'C' : {return IsMemberKamus(word,
P->NextC, nchar+1);}
            case 'D' : {return IsMemberKamus(word,
P->NextD, nchar+1);}
            case 'E' : {return IsMemberKamus(word,
P->NextE, nchar+1);}
            case 'F' : {return IsMemberKamus(word,
P->NextF, nchar+1);}
            case 'G' : {return IsMemberKamus(word,
P->NextG, nchar+1);}
            case 'H' : {return IsMemberKamus(word,
P->NextH, nchar+1);}
            case 'I' : {return IsMemberKamus(word,
P->NextI, nchar+1);}
            case 'J' : {return IsMemberKamus(word,
P->NextJ, nchar+1);}
            case 'K' : {return IsMemberKamus(word,
P->NextK, nchar+1);}
            case 'L' : {return IsMemberKamus(word,
P->NextL, nchar+1);}
            case 'M' : {return IsMemberKamus(word,
P->NextM, nchar+1);}
            case 'N' : {return IsMemberKamus(word,
P->NextN, nchar+1);}
            case 'O' : {return IsMemberKamus(word,
P->NextO, nchar+1);}
            case 'P' : {return IsMemberKamus(word,
P->NextP, nchar+1);}
            case 'Q' : {return IsMemberKamus(word,
```

```
        case 'R' : {return IsMemberKamus(word,
P->NextR, nchar+1);}
        case 'S' : {return IsMemberKamus(word,
P->NextS, nchar+1);}
        case 'T' : {return IsMemberKamus(word,
P->NextT, nchar+1);}
        case 'U' : {return IsMemberKamus(word,
P->NextU, nchar+1);}
        case 'V' : {return IsMemberKamus(word,
P->NextV, nchar+1);}
        case 'W' : {return IsMemberKamus(word,
P->NextW, nchar+1);}
        case 'X' : {return IsMemberKamus(word,
P->NextX, nchar+1);}
        case 'Y' : {return IsMemberKamus(word,
P->NextY, nchar+1);}
        case 'Z' : {return IsMemberKamus(word,
P->NextZ, nchar+1);}
```

IV. KEKURANGAN

Algoritma ini hanya efektif jika pencarian dilakukan pada data yang bersifat tetap. Dengan kata lain, akan efektif jika hanya dilakukan sekali pendataan atau pengondisian data dan pencarian pada data yang sama dilakukan berkali-kali. Hal ini dikarenakan proses pengondisian memiliki kompleksitas yang cukup besar.

V. KESIMPULAN

dalam beberapa kasus, modifikasi dari algoritma yang telah ada penting untuk meningkatkan efektifitas dari proses pencarian dan meminimalisis waktu pencarian dari suatu algoritma pencarian. Pada proses pencarian kata yang telah dibahas, dengan mengubah bentuk dari penyimpanan kamus, dapat dilakukan pencarian dengan menggunakan prinsip DFS sehingga didapat kompleksitas sebesar O(x) dengan x adalah panjang kata.

VII. UCAPAN TERIMA KASIH

Puji syukue penulis panjatkan kepada Tuhan YME karena atas isin-Nya makalah ini dapat selesai.. Penulis mengucapkakn terima kasih kepada bapak Rinaldi Munis atas bimbingannya selaku dosen matematika diskrit, serta kepada pendahulu-pendahlu yang telah memberikan karya-karya terkait algoritma penyelesaikan berbagai masalah terutama dalam masalah pencarian sehingga membantu dalam menyelesaikan makalah ini.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. *Diktat Kuliah IF2091 Matematika Diskrit*. Bandung : Penerbit informatika. 2008

[2] Happy Atrinawati, Lovita. *Analisis Kompleksitas Algoritma untuk Berbagai Macam Metode Pencarian Nilai(Searching) dan Pengurutan Nilai (Sorting) pada Tabel*. Bandung : ITB , 2005

[3] "[pdf] Algoritma Pencarian (Searching Algorithm) - Lecturer EEPIS" ~ lecturer.eepis-

[4] www.softpanorama.org

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, 12 Desmber 2014

Fikri Aulia 13513050