Penggunaan *BK-Tree* dalam *Spell-checker*

Vicko Novianto / 13513092
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13513092@std.stei.itb.ac.id

Abstrak*—*BK-Tree* adalah struktur data yang dibuat untuk mencari string yang ejaannya mirip dengan suatu string. *BK-Tree* banyak digunakan dalam *spell-checker*. *BK-Tree* menggunakan Jarak Levenshtein yaitu banyak penghapusan, penyisipan, dan penggantian minimum agar suatu string sama dengan string lainnya. Performansi *BK-Tree* makin baik jika banyak kata di kamus makin besar dan bergantung juga pada beberapa faktor lainnya. Performansi *BK-Tree* juga lebih baik daripada *linear search.

***Kata kunci*—Jarak Levenshtein, *BK-Tree*, *spell-checker*, *query*.**

I. PENDAHULUAN

Spell-checker atau dalam bahasa Indonesia disebut pemeriksa ejaan adalah program komputer yang mencari kesalahan ejaan dalam suatu teks dan memperbaikimya. Spell-checker bekerja dengan cara membandingkan kata-kata dalam suatu teks dengan kata-kata yang ejaannya sudah benar yang disimpan dalam basis data.[1]

Spell-checker banyak digunakan dalam aplikasi komputer, terutama aplikasi pengolah kata seperti Micorosft Word. Contoh aplikasi yang menggunakan *spell-checker* adalah Google Chrome, Microsoft Outlook, dan Google Translate. Contoh yang lain adalah ketika kita mencari suatu kata yang ejaannya salah dengan Google, maka kita akan diberi tahu kata yang ejaannya sudah benar.

Ada beberapa metode yang digunakan dalam merealisasikan *spell-checker* yaitu menggunakan *Hash, self-balancing Binary Search Tree, Skip List, Trie, Ternary Search Tree,* dan *BK-Tree*[2]. Pada makalah ini kita akan fokus pada pembahasan mengenai *BK-Tree.*

II. DASAR TEORI

BK-Tree atau *Burkhard-Keller Tree* adalah struktur data berupa pohon yang dibuat oleh *Burkhard* dan *Keller* pada tahun 1973 untuk mencari satu atau beberapa string yang mirip atau mendekati string yang menjadi masukan.[4] *BK-Tree* digunakan untuk melakukan *'fuzzy' search* dan *spell-checking.* Contohnya ketika kita mencari kata "thab" maka *spell-checker* akan menampilkan kata "than" dan "that".

Untuk mencari satu atau beberapa string yang mirip atau mendekati string yang menjadi masukan, dibutuhkan suatu metode yaitu dengan mencari *Levenshtein Distance* (Jarak Levenshtein). Jarak Levenshtein antara dua string adalah jumlah minimum penyisipan, penghapusan, dan penggantian yang dibutuhkan agar satu string sama dengan string lainnya. Contohnya, Jarak Levenshtein antara kata "buku" dan "kuku" adalah satu, karena hanya diperlukan satu kali penggantian huruf "b" dengan huruf "k" atau sebaliknya. Contoh lainnya, Jarak Levenshtein antara kata "kaki" dan "aki" adalah satu, karena hanya dipelrukan satu kali penghapusan huruf "k" pada kata "kaki" atau penambahan huruf "k" pada kata "aki".

Jarak Levenshtein ternyata membentuk sebuah ruang metrik. Ruang metrik adalah sebuah himpunan yang jarak antara anggotanya terdefinisi. Ruang metrik memiliki tiga sifat, yaitu :

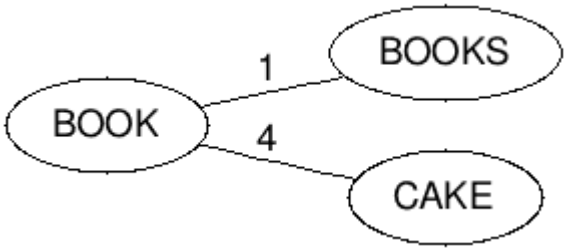
$$d(x,y) = 0 \leftrightarrow x = y$$
$$d(x,y) = d(y,x)$$
$$d(x,y) + d(y,z) \geq d(x,z)$$

dengan d(x,y) adalah jarak antara x dan y.

Jarak Levenshtein juga memenuhi *Triangle Inequality* (Pertidaksamaan Segitiga). Pertidaksamaan Segitiga menyatakan bahwa jumlah dari penjang dua sisi segitiga lebih besar daripada panjang satu sisi lainnya. Maka jarak dari x ke y tidak mungkin lebih panjang dari jarak x ke y melalui suatu titik perantara.

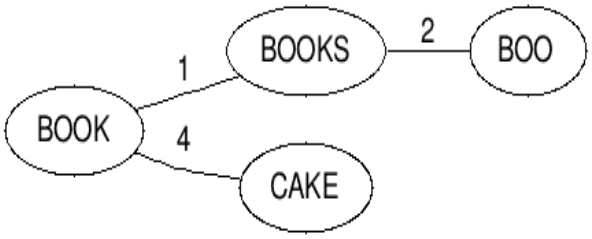
Asumsikan kita memiliki dua parameter yaitu *query*, yaitu string yang ejaannya salah, dan *n*, yaitu jarak Levenshtein maksimum antara *query* dan kata yang ejaannya benar yang masih bisa dianggap mendekati. Besaran n bebas ditentukan oleh kita. Lalu kita ambil sembarang string yaitu *test*, lalu kita bandingkan dengan *query.* Misalkan jarak Levenshtein antara *query* dan test adalah *d.* Karena pertidaksamaan segitiga berlaku, maka semua jarak Levenshtein yang dihasilkan maksimum bernilai *d+n* dan minimum bernilai *d-n*.

Setiap node dari *BK-Tree* bebas memiliki jumlah anak berapapun dan setiap sisi memiliki sebuah angka yang menyatakan jarak Levenshtein. Semua anak yang bersisian dengan sisi bernomor n memiliki jarak Levenshtein sebesar n dengan orang tuanya. Contohnya :



Gambar 1 : *Contoh BK-Tree sederhana*
Sumber:http://nullwords.files.wordpress.com/2013/03/bk1-e1363207059945.png

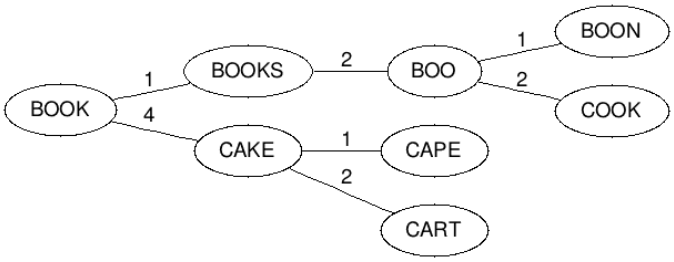
Untuk membangun *BK-Tree* dari sebuah kamus, ambil sebuah kata acak. Lalu jadikan kata acak tersebut sebagai akar. Jika kita ingin menambahkan suatu kata *K* pada *BK-Tree*, maka hitung dulu jarak Levenshtein antara *K* dengan akar dari *BK-Tree* yang kita buat. Misalkan jarak Levenshtein yang telah kita hitung adalah *n*. Lalu carilah sisi-sisi yang bersisian dengan akar yang bernomor n. Jika sisi yang bernomor n tidak ada, maka tambahkan simpul berlabel *K* sebagai anak dari akar. Setelah itu beri nomor sisi yang menghubungkan *K* dengan akar. Nomor yang diberikan menyatakan jarak Levenshtein antara *K* dengan akar. Setelah sisi-sisi bernomor n ditemukan, maka cek apakah simpul-simpul anak yang bersisian dengan sisi-sisi bernomor n tersebut merupakan daun atau tidak. Jika simpul-simpul tersebut merupakan daun, maka tambahkan simpul berlabel *K* sebagai anak dari daun tersebut dan beri nomor pada sisi yang menghubungkan *K* dengan orang tuanya dengan nomor tersebut menyatakan jarak Levenshtein antara *K* dengan orangtuanya. Jika simpul-simpul tersebut bukan merupakan daun, maka ulangi secara rekursif langkah-langkah di atas dengan menganggap simpul-simpul tersebut sebagai akar.



Gambar 2 *: BK-Tree setelah kata "BOO" ditambahkan*
Sumber:http://nullwords.files.wordpress.com/2013/03/bk2-e1363207047338.png

Misalkan kita memiliki *BK-Tree* seperti Gambar 1. Lalu kita ingin menambahkan kata "BOO" pada *BK-Tree* tersebut. Pertama-tama kita menghitung jarak Levenshtein antara "BOO" dengan akar yaitu kata "BOOK". Jarak Levenshtein yang telah dihitung adalah satu. Lalu kita cari sisi yang bernomor 1. Sisi bernomor 1 ditemukan dan kita cek apakah simpul anak yang bersisian dengan sisi bernomor 1 yaitu "BOOKS" merupakan daun atau tidak. Ternyata simpul "BOOKS" merupakan daun. Maka kita tambahkan "BOO" sebagai simpul anak dari "BOOKS". Setelah itu kita hitung jarak Levenshtein antara "BOO" dengan "BOOKS" yaitu 2. Selanjutnya kita beri nomor 2 pada sisi yang menghubungkan "BOO" dengan "BOOKS".

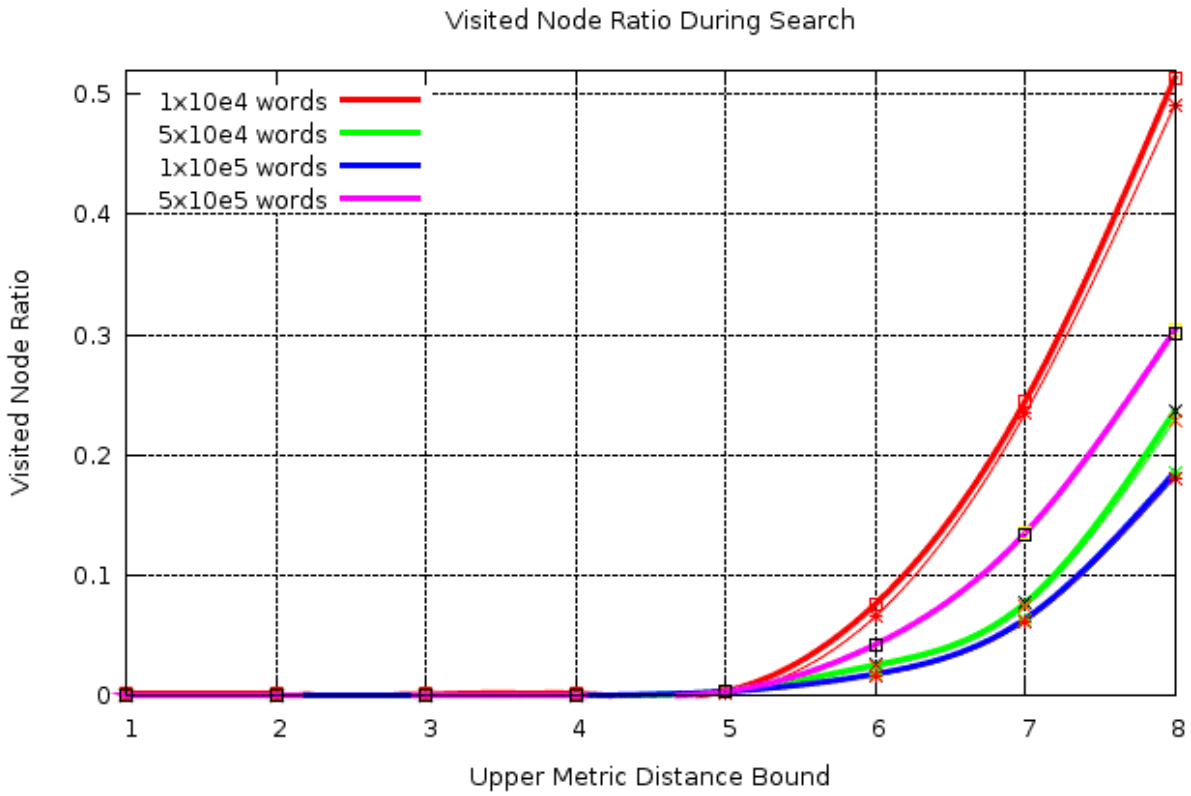
Untuk melakukan pencarian suatu string dalam *BK-Tree*, misalkan string *query*, hitung jarak Levenshtein antara *query* dengan akar. Jika jarak Levenshtein tersebut lebih kecil sama dengan *n*, maka tambahkan akar sebagai salah satu jawaban. Jika jarak Levenshtein tersebut lebih besar dari *n*, maka abaikan akar tersebut. Lalu tentukan *n*, yaitu jarak Levenshtein maksimum antara *query* dan kata yang ejaannya benar yang masih bisa dianggap mendekati. Besaran n bebas ditentukan oleh kita. Misalkan jarak Levenshtein yang dihasilkan adalah *d*. Lalu cek sisi-sisi yang memiliki nomor lebih besar sama dengan *d-n* dan lebih kecil sama dengan *d+n*. Setelah itu cek simpul-simpul anak yang bersisian dengan sisi-sisi tersebut. Lalu hitung jarak Levenshtein antara simpul-simpul anak dengan *query*. Jika jarak Levenshtein tersebut lebih kecil sama dengan *n*, maka tambahkan simpul-simpul anak sebagai salah satu jawaban. Jika jarak Levenshtein tersebut lebih besar dari *n*, maka abaikan simpul-simpul tersebut. Selanjutnya cek apakah simpul-simpul anak tersebut merupakan daun atau tidak. Jika simpul-simpul tersebut daun, maka proses berhenti hingga semua daun diperiksa. Jika simpul-simpul tersebut bukan daun, maka ulangi proses di atas secara reskursif dengan simpul-simpul tersebut dianggap sebagai akar.



Gambar 3 *: BK-Tree yang sudah ditambahkan beberapa kata*
Sumber:http://nullwords.files.wordpress.com/2013/03/bk31-e1363207034407.png

Pertama-tama, misalkan *query* adalah "CAQE" dan *n* adalah 1. Lalu hitung jarak Levenshtein antara "BOOK" dengan "CAQE" yaitu *d* yang bernilai 4. Karena 4 tidak lebih kecil sama dengan *n* yaitu 1, maka jangan tambahkan "BOOK" sebagai jawaban. Lalu telusuri sisi-sisi yang bernomor di antara 3 (4-1) dan 5(4+1) inklusif. Lalu kita telusuri sisi bernomor 4. Hitung *d* yang baru yaitu jarak Levenshtein antara "CAKE" dan "CAQE" yaitu 1. Karena *d* <= *n* (1 <= 1), maka tambahkan "CAKE" sebagai jawaban. Lalu telusuri sisi bernomor di antara 0 (1-1) dan 2 (1+1). Lalu kita telusuri sisi bernomor 1. Hitung jarak Levenshtein antara "CAPE" dan "CAQE" yaitu 1. Karena *d* <= *n* (1 <= 1), maka tambahkan "CAPE" sebagai jawaban. Karena "CAPE" adalah simpul daun, maka proses berhenti. Lalu kita telusuri sisi bernomor 2. Kita lalu menghitung jarak Levenshtein antara "CART" dan "CAQE" yaitu 2. Karena *d* > *n* (2 > 1), maka jangan tambahkan "CART" sebagai jawaban. Karena "CART" adalah daun, maka proses berhenti.

III. PERFORMANSI *BK-TREE*



***Gambar 4** : Grafik perbandingan antara rasio node yang dikunjungi dengan besar toleransi (n)*

Sumber: https://camo.githubusercontent.com/f91ad36c3fefdbb9bed123fea07ebf4f6c2c94f7/68747470733a2f2f7261772e6769746875622e636f6d2f76792f626b2d747265652f6d61737465722f524541444d452e7265706f72742e706e67

Grafik di atas adalah perbandingan antara rasio node yang dikunjungi dengan besar toleransi (*n*). Rasio node yang dikunjungi adalah perbandingan jumlah node yang ditemukan dengan jumlah node yang ditelusuri. Makin besar rasio node yang dikunjungi, maka makin banyak proses pencarian yang *redundant* (tidak perlu) karena node yang ditemukan makin sedikit. Semakin besar rasio node yang dikunjungi, maka akan semakin lama juga waktu yang dibutuhkan untuk menelusuri node-nodenya karena banyaknya tahapan yang diperlukan juga meningkat. Besar toleransi adalah jarak Levenshtein maksimum antara *query* dan kata yang ejaannya benar yang masih bisa dianggap mendekati.

Dari grafik di atas dapat diambil beberapa kesimpulan. Pertama, semakin besar besar toleransi maka semakin banyak node yang harus ditelusuri dan rasio node yang ditkunjungi semakin besar. Kedua, *BK-Tree* memiliki performansi yang sangat baik ketika besar toleransi lebih kecil sama dengan lima. Keitga, semakin besar ukuran kamusnya, maka semakin baik pula performansi *BK-Tree*. Keempat, performansi *BK-Tree* yang paling buruk terjadi ketika *n = 8* dan jumlah kata dalam kamus = 10000.

Tabel I Performansi BK-Tree

Ukuran kamus (kata)	Besar toleransi (*n*)	Simpul yang ditelusuri (%)	Simpul yang ditemukan (%)
10000	1	0,110	0,0100
	2	0,110	0,0100
	3	0,110	0,0100
	4	0,160	0,0300
	5	0,370	0,1100
	6	7,600	6,5800
	7	24,460	23,4300
	8	51,360	49,0900
50000	1	0,0022	0,0020
	2	0,0023	0,0021
	3	0,0251	0,0030
	4	0,0408	0,0127
	5	0,3943	0,3196
	6	2,5430	2,3869
	7	7,6876	7,3874
	8	23,6635	22,9339
100000	1	0,0012	0,0010
	2	0,0012	0,0011
	3	0,0013	0,0017
	4	0,0231	0,0085
	5	0,3383	0,2998
	6	1,7957	1,7079
	7	6,3571	6,1654

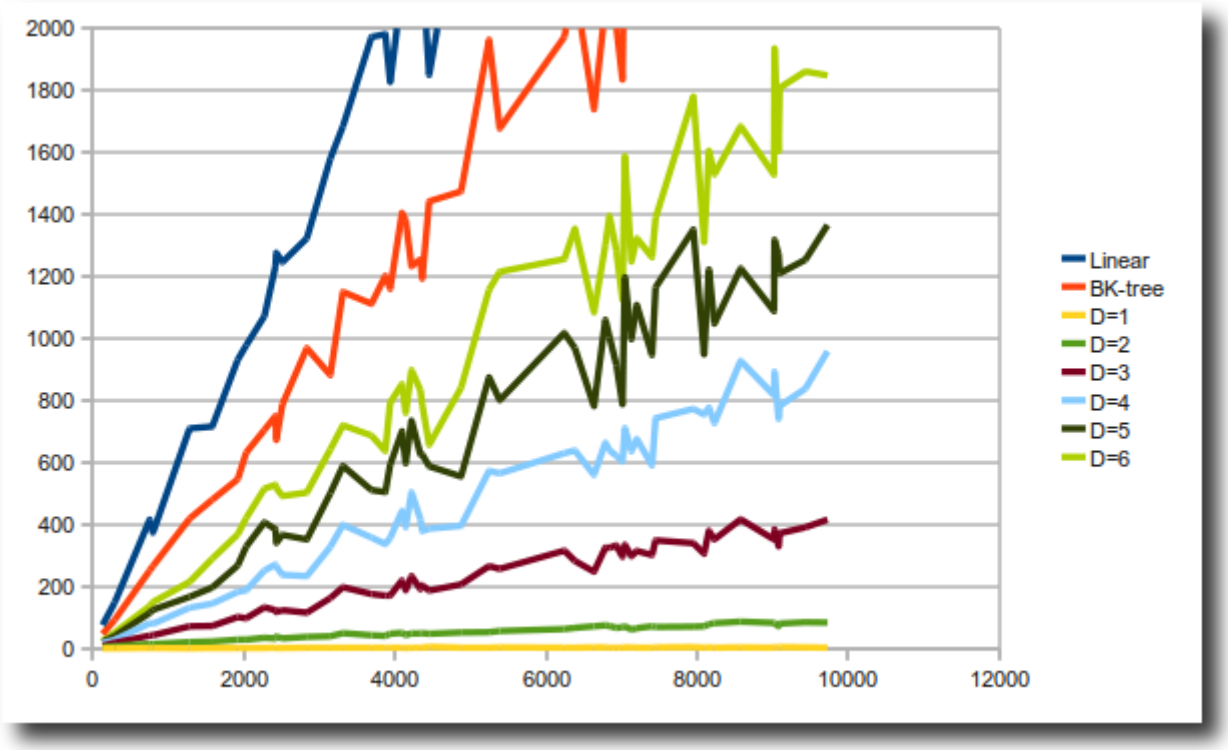
	8	18,5599	18,0996
500000	1	0,0027	0,0002
	2	0,0029	0,0002
	3	0,0039	0,0011
	4	0,0012	0,0081
	5	0,3444	0,3213
	6	0,4244	0,4201
	7	13,4834	13,3619
	8	30,3728	30,1685

Sumber : https://github.com/vy/bk-tree

Tabel I di atas menggambarkan performansi *BK-Tree*. Di dalam setiap tes, 100 kata acak dicari di dalam basis data yang juga dibuat secara random. Kata-kata yang disimpan di basis data memiliki panjang dari 5 sampai 10 karakter.

V. KESIMPULAN

Spell-checker sudah menjadi fitur umum yang dimiliki banyak aplikasi pemroses kata, bahkan aplikasi pengirim *e-mail*, *browser*, dan *translator*. Banyak metode yang bisa digunakan dalam *spell-checking*, salah satunya menggunakan *BK-Tree*. *BK-Tree* adalah struktur data yang didesain khusus untuk mencari string yang ejaannya "mirip" atau mendekati jata-kata yang terdefinisi di dalam kamus. *BK-Tree* cukup mudah untuk diimplementasikan karena operasinya menyangkut operasi rekursif. Performansi *BK-Tree* pun lebih baik daripada *linear search* dan sebenarnya tergantung dari besar toleransi yang dipilih. Performansi *BK-Tree* juga lebih baik ketika jumlah kata di kamus sangat besar. Oleh karena itu, *BK-Tree* cocok dijadikan struktur data *Spell-checker*.



***Gambar 5** : Performansi BK-Tree dengan D (besar toleransi) beragam dengan linear search tanpa BK-Tree*
Sumber : http://www.kafsemo.org/2010/08/03_varying-maximum-distance.png

Gambar 5 menunjukkan performansi *BK-Tree* dengan D (besar toleransi) bervariasi dari 1 sampai 6, *BK-Tree* dengan D > 6, dan *linear search* tanpa *BK-Tree*. *BK-Tree* dengan D > 6 ditunjukkan dengan garis berwarna oranye. Dari gambar tersebut dapat disimpulkan bahwa semakin kecil D, maka semakin cepat waktu prosesnya. Jika D > 6, maka performansi *BK-Tree* hampir sama dengan *linear search*. Oleh karena itu, jika menggunakan *BK-Tree*, lebih baik membatasi D agar tidak terlalu besar sehingga program bisa lebih cepat dalam mencari string. Namun bagaimanapun juga *Kk-Tree* tidak pernah lebh lambat daripada *linear search* biasa.

REFERENSI

[1] http://www.merriam-webster.com/dictionary/spell-checker, 11 Desember 2014, 07:06
[2] http://www.geeksforgeeks.org/data-structure-dictionary-spell-checker/, 11 Desember 2014, 07:12
[3] http://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/, 11 Desember 2014, 07:19
[4] http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-Trees, 11 Desember 2014, 07:27
[5] http://www-history.mcs.st-and.ac.uk/~john/MT4522/Lectures/L5.html, 11 Desember 2014, 07:39
[6] http://mathworld.wolfram.com/TriangleInequality.html, 11 Desember 2014, 08:03
[7] https://github.com/vy/bk-tree, 11 Desember 2014, 10:12

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, 11 Desember 2014

Vicko Novianto / 13513092