Penerapan Algoritma Runut-balik dalam Fungsi *Hash*

Puthut Prabancono-13506068

Institut Teknologi Bandung
Jalan Ganeca 10, Bandung
e-mail: puthut_aja@yahoo.com

ABSTRAK

Dalam ilmu komputer dikenal sebuah algoritma pencarian yang menerapkan *depth first search* yang dikenal dengan nama algoritma runut-balik (*backtracking*). Algoritma ini banyak digunakan pada program *games* dan juga intelegensia buatan. Algoritma runut-balik ternyata dapat juga digunakan pada fungsi *hash*, yaitu pada penanganan bentrokan yang terjadi jika beberapa *record* ternyata menghasilkan nilai yang sama. Makalah ini membahas bagaimana sebuah fungsi *hash* dapat digabungkan dengan algoritma runut-balik sehingga menghasilkan sebuah fungsi *hash* yang efektif dan efisien.

Kata kunci: *Backtracking, hash,* runut-balik, Cichelli.

1. PENDAHULUAN

1.1 Algoritma runut-balik

Algoritma runut-balik atau *Backtracking* adalah algoritma yang berbasis pada DFS untuk mencari solusi persoalan yang lebih mangkus. Runut-balik, yang merupakan perbaikan dari algoritma *Brute force*, secara sistematis mencari solusi persoalan di antara semua kemungkinan solusi yang ada. Dengan metode runut-balik, kita tidak perlu memeriksa semua kemungkinan solusi yang ada. Hanya pencarian yang mengarah ke solusi saja yang selalu dipertimbangkan. Akibatnya, waktu pencarian dapat dihemat. Runut-balik lebih alami dinyatakan dalam algoritma rekursif. Kadang-kadang disebutkan pula bahwa runut-balik merupakan bentuk tipikal dari algoritma rekursif.

Istilah runut-balik pertama kali deperkenalkan oleh D.H. Lehmer pada tahun 1950. Selanjutnya R.J. Walker, Golomb, dan Baumert menyajikan uraian umum tentang runut-balik dan penerapannya pada berbagai persoalan. Saat ini algoritma runut-balik banyak diterapkan untuk program *games* (seperti permainan *tic-tac-toe*, menemukan jalan keluar dalam sebuah labirin, catur, dll) dan masalah-masalah pada bidang kecerdasan buatan (*artificial intelligence*).

Untuk menerapkan metode runut-balik, properti berikut didefinisikan:

1. Solusi persoalan.
   - Solusi dinyatakan sebagai vektor dengan *n-tuple*: $X = (x1, x2, \ldots, xn)$, $xi \in Si$.
   - Mungkin saja $S1 = S2 = \ldots = Sn$.
   - Contoh: $Si = \{0, 1\}$, $xi = 0$ atau 1

2. Fungsi pembangkit nilai *xk*
   Dinyatakan sebagai:
   $T(k)$
   $T(k)$ membangkitkan nilai untuk *xk*, yang merupakan komponen vektor solusi.
3. Fungsi pembatas (pada beberapa persoalan fungsi ini dinamakan fungsi kriteria)
   - Dinyatakan sebagai
     $B(x1, x2, \ldots, xk)$
   - *B* bernilai *true* jika $(x1, x2, \ldots, xk)$ mengarah ke solusi. Jika *true*, maka pembangkitan nilai untuk $xk+1$ dilanjutkan, tetapi jika *false*, maka $(x1, x2, \ldots, xk)$ dibuang dan tidak dipertimbangkan lagi dalam pencarian solusi.
   - Fungsi pembatas tidak selalu dinyatakan sebagai fungsi matematis. Ia dapat dinyatakan sebagai predikat yang bernilai *true* atau *false*, atau dalam bentuk lain yang ekivalen.

Semua kemungkinan solusi dari persoalan disebut **ruang solusi** (*solution space*). Jika $xi \in Si$, maka $S1 \times S2 \times \ldots \times Sn$ disebut **ruang solusi**. Jumlah anggota di dalam ruang solusi adalah $| S1| \cdot | S2| \cdot \ldots \cdot | Sn |$.

Algoritma runut-balik memperbaiki pencarian solusi secara *exhaustive search* dengan mencari solusi secara sistematis. Untuk memfasilitasi pencarian ini, maka ruang solusi diorganisasikan ke dalam struktur pohon. Tiap simpul pohon menyatakan status (state) persoalah, sedangkan sisi (cabang) dilabeli dengan nilai-nilai x_i. Lintasan dari akar ke daun menyatakan solusi yang mungkin. Seluruh lintasan dari akar ke daun membentuk ruang solusi. Pengorganisasian pohon ruang solusi diacu sebagai pohon ruang status (*state space tree*).

Langkah-langkah pencarian solusi pada pohon ruang status yang dibangun secara dinamis:

- Solusi dicari dengan membentuk lintasan dari akar ke daun. Aturan pembentukan yang dipakai adalah mengikuti aturan pencarian mendalam

(DFS). Simpul-simpul yang sudah dilahirkan dinamakan **simpul hidup** (*live node*). Simpul hidup yang *sedang* diperluas dinamakan **simpul-E** (*Expand-node*).

- Tiap kali simpul-E diperluas, lintasan yang dibangun olehnya bertambah panjang. Jika lintasan yang sedang dibentuk tidak mengarah ke solusi, maka simpul-E tersebut "dibunuh" sehingga menjadi simpul mati (dead node). Fungsi yang digunakan untuk membunuh simpul-E adalah dengan menerapkan fungsi pembatas (bounding function). Simpul yang sudah mati tidak akan pernah diperluas lagi.
- Jika pembentukan lintasan berakhir dengan simpul mati, maka proses pencarian diteruskan dengan membangkitkan simpul anak yang lainnya. Bila tidak ada lagi simpul anak yang dapat dibangkitkan, maka pencarian solusi dilanjutkan dengan melakukan runut-balik ke simpul hidup terdekat (simpul orangtua). Selanjutnya simpul ini menjadi simpul-E yang baru.
- Pencarian dihentikan bila kita telah menemukan solusi atau tidak ada lagi simpul hidup untuk runut-balik.

1.2 Fungsi *Hash*

Hashing adalah transformasi aritmatik sebuah string dari karakter menjadi nilai yang merepresentasikan string aslinya. Menurut bahasanya, *hash* berarti memenggal dan kemudian menggabungkan.

Hashing digunakan sebagai metode untuk menyimpan data dalam sebuah array agar penyimpanan data, pencarian data, penambahan data, dan penghapusan data dapat dilakukan dengan cepat. Ide dasarnya adalah menghitung posisi *record* yang dicari dalam array, bukan membandingkan *record* dengan isi pada array. Fungsi yang mengembalikan nilai atau kunci disebut fungsi *hash* (*hash* function) dan array yang digunakan disebut tabel *hash* (*hash* table). Secara teori, kompleksitas waktu (T(n)) dari fungsi *hash* yang ideal adalah O(1). Untuk mencapai itu setiap *record* membutuhkan suatu kunci yang unik.

Fungsi *hash* menyimpan nilai asli atau kunci pada alamat yang sama dengan nilai *hash*nya. Pada pencarian suatu nilai pada tabel *hash*, yang pertama dilakukan adalah menghitung nilai *hash* dari kunci atau nilai aslinya, kemudian membandingkan kunci atuau nilai asli dengan isi pada memori yang beralamat nomor *hash*nya. Dengan cara ini, pencarian suatu nilai dapat dilakukan dengan cepat tanpa harus memeriksa seluruh isi tabel satu per satu.

Selain digunakan pada penyimpanan data, fungsi *hash* juga digunakan pada algoritma enkripsi sidik jari digital (*fingerprint*) untuk mengautentifikasi pengirim dan penerima pesan. Sidik jari digital diperoleh dengan fungsi *hash*, kemudian nilai *hash* dan tanda pesan yang asli dikirim kepada penerima pesan. Dengan menggunakan fungsi *hash* yang sama dengan pengirim pesan, penerima pesan mentransformasikan pesan yang diterima. Nilai *hash* yang diperoleh oleh penerima pesan kemudian dibandingkan dengan nilai *hash* yang dikirim pengirim pesan. Kedua nilai *hash* harus sama, jika tidak, pasti ada masalah.

Hashing selalu merupakan fungsi satu arah. Fungsi *hash* yang ideal tidak bisa diperoleh dengan melakukan *reverse engineering* dengan menganalisa nilai *hash*. Fungsi *hash* yang ideal juga seharusnya tidak menghasilkan nilai *hash* yang sama dari beberapa nilai yang berbeda. Jika hal yang seperti ini terjadi, inilah yang disebut dengan bentrokan (*collision*). Kemungkinan terjadinya bentrokan tidak dapat dihindari seratus persen. Fungsi *hash* yang baik dapat meminimalkan kemungkinan terjadinya bentrokan.

2. FUNGSI *HASH* SEMPURNA MINIMAL

Dengan fungsi *hash* yang sempurna, dapat diperolah *record* pada tempat yang tepat, tanpa perlu khawatir akan adanya bentrokan. Kasus spesial dari fungsi *hash* sempurna adalah fungsi *hash* sempurna minimal. Fungsi *hash* sempurna minimal memenuhi kriteria performa yang cepat dan penggunaan memori yang sedikit. Namun, untuk menemukan fungsi *hash* yang tepat sangat susah.

Sebuah fungsi *hash* sempurna diperkenalkan oleh R. Cichelli dalam bukunya pada tahun 1980. Cichelli menyarankan penggunaan *exhaustive search* untuk menemukan fungsi tersebut, dan mendiskusikan beberapa heuristik untuk mempercepat pencarian.

Fungsi *hash* Cichelli sangat sederhana, misalkan diberikan sebuah kata w, nilai *hash*-nya adalah Value(First_Letter(*w*)) + Value(Last_Letter(*w*)) + Length(*w*). Pemetaan fungsi Value() biasanya diimplementasikan sebagai sebuah table yang memetakan karakter menjadi nomor. Pada Tabel 1 dapat dilihat contoh tabel dan nilai *hash* dari sebuah fungsi *hash* sempurnaa minimal untuk hari-hari dalam satu minggu. Dapat dilihat bahwa tabel yang diperlukan tidak terlalu besar, hal ini disebabkan karena hanya huruf pertama dan terakhir yang perlu dipetakan. Menghitung nilai *hash* hanya memerlukan dua akses pada tabel dan dua penambahan.

Tabel 1 Tabel *hash* berisi nama-nama hari

Kata	*Hash*
Sunday	0
Monday	5
Tuesday	3
Wednesday	6
Thursday	4
Friday	1

Saturday	2

Kata	***Hash***
f	-8
m	-9
s	-4
t	-7
w	-6
y	3

Algoritma Cichelli yang sebenarnya menggunakan pendekatan exhaustive dengan backtracking. Setiap pemetaan yang mungkin dicoba di cari dangan Value(), dan setiap penugasan menghasilkan benturan, alur pencarian dihentikan, dan solusi lain dicoba. Algoritma ini mudah untuk dikodekan, tapi proses penghitungannya bisa menghasilkan waktu yang bertambah secara eksponensial. Cichelli menyaran kan dua cara untuk mempercepat pendekatan ini. Cara pertama adalah dengan mengurutkan kuncinya: menghitung frekuensi huruf pertama dan terakhir, dan kemudian mengurutkan kata-kata berdasarkan jumlah huruf pertama dan terakhir. Cara ini akan menjamin bahwa huruf yang paling sering muncul diberikan nilai terlebih dahulu, dan akan meningkatkan kemungkinan terjadinya bentrokan ditemukan saat kedalaman pohon masih belum terlalu dalam. Hal ini akan mengurangi ongkos runut-balik.

Walaupun demikian, cara ini tidak selalu efektif. Misal sebuah himpunan kunci:

{aa, ab, ac, ad, ae, af, ag, bd, cd, ce, de, fg},
yang apabila diurutkan akan menjadi
{aa, ad, ac, ae, ab, af, ag, cd, de, bd, ce, fg}.

Sekaran, nilai dari kunci cd ditentukan segera setelah ac diberikan nilai, tapi tes benturan pada cd tidak dilakukan sampai lima cabang lagi dalam pohon pencarian diambil. Ini akan membuat ongkos pada benturan pada cd menjadi jauh lebih mahal dari yang seharusnya.

Cichelli menyarankan heuristik kedua untuk mengatasi masalah ini. Jika sebuah kunci dalam list yang sudah terurut ini memiliki huruf yang sudah pernah muncul, kunci tersebut diletakkan tepat setelah kunci yang melengkapi pemetaannya. Pada contoh sebelumnya, cd akan dipindahkan tepat setelah ac. Dalam melakukan hal tersebut, harus diusahakan untuk menjaga urutan yang sebenarnya.

(bisa ditambahkan implementasinya dalam bahasa C++)

Ada lima langkah dalam algoritma Cichelli:

1. Menghitung frekuensi huruf pertama dan terakhir
2. Mengurutkan kunci dalam urutan menurun dalam jumlah frekuensi
3. Atur kembali urutan dari awal list, sehingga jika huruf pertama dan terakhir dari sebuah kunci sudah pernah muncul, kunci itu diletakkan tepat setelah kunci yang melengkapi.
4. Proses pencarian: untuk setiap kunci, jika kedua nilai dari huruf pertama dan terakhir sudah ditugaskan, letakkan kunci di dalam tabel (langkah ke 5); selain itu, jika hanya satu nilai yang perlu ditugaskan, coba seluruh penugasan yang mungkin. Jika kedua nilai harus ditugaskan, gunakan dua kalang bersarang untuk mencoba seluruh penugasan yang mungkin untuk kedua huruf.
5. Setelah ada kunci, panjang, dan nilai pemetaan, hitung alamatnya di dalam tabel. Runut-balik ke langkah 4 jika terjadi bentrokan; jika tidak, masukkan ke tabel. Jika huruf terakhir sudah dimasukkan, sebuah *hash* sempurna sudah ditemukan; jika tidak, secara rekursif kembali ke langkah 4 untuk kunci berikutnya.

Ukuran tabel tidak pasti. Jika ukuran tabel sama dengan jumlah kunci, maka disebut *hash*ing sempurna minimal; jika tidak, beberapa kolom tetap kosong di akhir proses.

Cichelli tidak menyediakan cara untuk membatasi rentang nilai untuk mencoba untuk setiap huruf. Rentang ini memainkan peran yang sangat penting dalam algoritma ini. Rentang yang kecil bisa menghasilkan tidak ada solusi, dan rentang yang terlalu besar dapat membutuhkan pohon pencarian yang terlalu lebar.

Sudah ditemukan aturan umum yang efektif untuk hampir semua himpunan, yang menghasilkan rentang kecil yang memiliki solusi. Karena pertambahan panjang kata dan indeks yang ingin dihasilkan kecil (termasuk nol), sebagian besar huruf akan ditugaskan menjadi nilai negatif. Selain itu, nilai dari huruf tidak akan menjadi sangat besar karena ukuran tabel adalah batas atas indeks.

Rentang yang dipilih adalah [-*MaxLen*, *TableSize* – *MinLen*], dengan *MaxLen* adalah panjang maksimum dari kunci yang ada di himpunan; *MinLen* adalah panjang minimum dari kunci; dan *TableSize* adalah ukuran dari tabel *hash*. Rentang biasanya kecil (untuk kunci pada tabel 1, anggap sebuah *hash* minimal, rentangnya adalah [-9,1]) tetapi masih tetap efektif. Hal ini bisa ditingkatkan lebih jauh dengan mengadaptasikan rentang secara dinamis. Misalkan terjadi ketika satu dari dua huruf sudah ditugaskan dengan suatu nilai. Ini terjadi dua kali pada algoritma: ketika hanya satu nilai harus ditugaskan dan di bagian loop terdalam dari loop bersarang. Pada kasus ini, nilai (misalkan v) dan panjang kata sudah diketahui, maka rentang bisa dihitung kembali dan menjadi [-*v* - *Len*, *TableSize* - *v* - *Len*], dengan *Len* adalah panjang kunci yang akan ditugaskan dengan indeks. Optimalisasi ini sangat meningkatkan kinerja dari algoritma, peningkatan kecepatan bisa terjadi. Rentang dinamis kadang-kadang bisa melebihi rentang aslinya, contoh, pada tabel 1, y ditugaskan dengan nilai 3, di luar rentang statis. Optimalisasi lebih lanjut dari rentang dinamis dapat diperoleh dengan mempertimbangkan kolom mana yang kosong. Jika mengikuti kolom pertama dan terakhir yang kosong, rentang dinamis bisa dibatasi sampai [*FirstFree* -*v* - *Len*, *LastFree* - *v* - *Len*].

Hasilnya adalah algoritma yang cukup cepat: sebuah *hash* sempurna minimal untuk 32 kata kunci dari ANSI C, (lihat tabel 2) bisa ditemukan dalam kurang dari satu detik dengan Pentium/90 dan 46 kata kunci yang tanpa bentrokan membutuhkan sekitar 4 detik.

Tabel 2. Nilai masing-masing huruf dan kata dengan fungsi *hash*

Huruf	Nilai	Huruf	Nilai
a	-5	l	29
b	-8	m	-4
c	-7	n	-1
d	-10	o	22
e	4	r	5
f	12	s	7
g	-7	t	10
h	4	u	26
i	2	v	19
k	31	w	2

Kata	*Hash*	Kata	*Hash*
auto	21	int	15
break	28	long	26
case	1	register	18
char	2	return	10
const	8	short	22
continue	5	signed	3
default	7	sizeof	25
do	14	static	6
double	0	struct	23
else	12	switch	17
enum	4	typedef	29
extern	9	union	30
float	27	unsigned	24
for	20	void	13
goto	19	volatile	31
if	16	while	11

4. KESIMPULAN

Penggunaan algoritma runut-balik ternyata bisa juga diaplikasikan dalam fungsi *hash*. Algoritma runut-balik digunakan dalam fungsi *hash* pada proses penentuan suatu *record* akan diletakkan di kolom mana pada tabel *hash* jika terjadi bentrokan.

Fungsi *hash* yang baik adalah fungsi *hash* yang sangat jarang menghasilkan bentrokan, semakin jarang semakin baik. Algoritma runut-balik digunakan sebagai salah satu solusi penanganan jika terjadi bentrokan.

REFERENSI

[1] Munir,Rinaldi, “Strategi Algoritmik”, ITB, 2006.

[2] http://www.eptacom.net/pubblicazion/pub/eng/mp*hash*.html Tanggal akses 16 Mei 2008