Algoritma Pencarian String dengan *Regular Expression*

Kamal Mahmudi

Mahasiswa Jurusan Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Labtek V, Jalan Ganeça 10 Bandung
email: if17111@students.if.itb.ac.id

ABSTRAK

Algoritma pencarian string atau juga biasa disebut sebagai algoritma pencocokan string merupakan algoritma yang berusaha menemukan tempat satu atau beberapa string (pola) di dalam sebuah string besar (teks).
Dalam melakukan pencarian, dapat dilakukan berbagai pendekatan. Masing-masing pendekatan memiliki kelebihan dan kekurangan tersendiri bergantung pada pola dan teks yang digunakan. Tulisan ini akan membahas algoritma pencarian string dengan pola berupa regular expression.
Kata kunci: algoritma pencarian string, *regular expression*, *finite automata*.

1. PENDAHULUAN

Secara alami, manusia selalu ingin menikmati kemudahan dalam menjalani kehidupan. Manusia tidak ingin direpotkan dengan hal-hal yang dapat dikerjakan selain dirinya. Ambil sebuah contoh, di tengah meningkatnya populasi di dunia, semakin meningkat pula jumlah benda yang ada. Tentu saja manusia tidak mau menghabiskan waktu untuk mencari salah satu benda yang diinginkan di antara banyaknya benda lain.

Dalam dunia digital, kumpulan benda tersebut dapat berupa kumpulan berkas di *harddisk*, internet, teks digital, dan sebagainya. Sementara benda yang dicari dapat berupa sebuah berkas, halaman web, bagian tertentu dari teks digital, dan sebagainya.

Manusia tidak lagi melakukan pencarian secara manual, sebagai gantinya mesin bekerja untuk mencapai apa yang diinginkan manusia. Mesin pencari berkembang di sistem operasi, internet, bahkan editor teks. Mesin melakukan pencarian dengan mencocokkan pola yang diberikan dengan tabel indeks (teks) yang tersedia.

Berbagai pendekatan dalam melakukan pencarian telah dikembangkan menyesuaikan pola dan teks yang berbeda-beda. Kebutuhan manusia yang bersifat dinamis mengharuskan kemampuan pencarian mesin pun bersifat dinamis. Hal ini dikarenakan manusia juga melakukan pencarian dengan format tertentu seperti hanya mencari kata "tangan" yang dilanjutkan dengan kata "kanan" atau "kiri". Dengan r*egular* expression, mesin memiliki kemampuan untuk melakukannya.

2. REGULAR EXPRESSION

Regular expression merupakan notasi yang digunakan untuk mendeskripsikan himpunan karakter string. Sebagai contoh, himpunan yang terdiri dari string "*Handel*", "*Händel*", dan "*Haendel*" dapat dideskripsikan dengan "H(ä|ae?)ndel" (atau alternatif lainnya, notasi ini menunjukkan bahwa notasi tersebut sesuai dengan setiap string yang diberikan).

2. 1. Definisi

Regular expression didefinisikan berdasarkan aturan teori bahasa formal. *Regular expression* terdiri dari konstanta dan operator yang menunjukkan himpunan-himpunan string dan operasi antar himpunan string tersebut secara berurutan.

Konstanta yang telah didefinisikan adalah:

- Himpunan kosong, diberi notasi ∅.
- String kosong, diberi notasi ε.
- Karakter, diberi notasi sesuai dengan karakter bahasa yang digunakan.

Operator yang telah didefinisikan adalah:

- Konkatenasi, misal {"ab", "c"} {"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- Alternasi, misal {"ab", "c"} | {"ab", "d", "ef"} = {"ab", "c", "d", "ef"}.
- *Kleene star*, menunjukkan semua himpunan yang dapat dibuat dengan melakukan konkatenasi 0 atau lebih banyak string dari string yang dilakukan

operasi ini. Misal {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.

Untuk mengurangi jumlah tanda kurung, diasumsikan *Kleene star* memiliki prioritas terbesar dilanjutkan dengan operasi konkatenasi baru operasi alternasi.

2. 2. POSIX (Portable Operating System Interface [for Unix])

Unix tradisional memiliki *regular expression* yang mengikuti konvensi umum namun sering kali berbeda antar kakas. Standar IEEE POSIX *Basic Regular Expressions* atau BRE (sementara itu dirilis pula alternatif yang disebut *Extended Regular Expressions* atau ERE) dirancang sedemikian rupa sehingga kompatibel terhadap tradisional (*Simple Regular Expression* atau SRE) namun juga menyediakan standar umum *regular expression* yang dimiliki kebanyakan kakas di unix.

Pada sintaks BRE, hampir semua karakter diberlakukan secara harfiah. Pengecualian diberlakukan untuk beberapa karakter berikut yang biasa disebut *metacharacter* atau *metasequence*.

Tabel 1 Daftar *Metacharacter*

Metacharacter	Deskripsi
`.`	Sesuai dengan satu karakter apa saja. Dalam kurung siku karakter titik sesuai dengan titik secara harfiah. Misal, `a.c` matches "*abc*", etc., but `[a.c]` matches only "*a*", ".", atau "*c*".
`[]`	Ekspresi kurung siku. Sesuai dengan satu karakter yang ada dalam kurung. Misal, `[abc]` sesuai "*a*", "*b*", atau "*c*". `[a-z]` menspesifikasikan sebuah *range* yang sesuai dengan huruf kecil dari "*a*" sampai "*z*". Format ini dapat dicampur: `[abcx-z]` sesuai dengan "*a*", "*b*", "*c*", "*x*", "*y*", or "*z*", sebagaimana `[a-cx-z]`. Karakter − diberlakukan secara harfiah jika berada di awal atau di akhir tanda kurung siku, atau jika diawali dengan karakter *escape* seperti: `[abc-]`, `[-abc]`, or `[a\-bc]`.
`[^]`	Sesuai dengan satu karakter apa saja yang tidak ada dalam kurung.
`^`	Sesuai dengan awal string. Pada kakas *line-based*, notasi ini sesuai dengan awal baris di mana saja.
`$`	Sesuai dengan akhir string. Pada kakas *line-based*, notasi ini sesuai dengan akhir baris di mana saja.
BRE: `\(\)` ERE: `()`	Mendefinisikan *subexpression* yang dapat dipanggil kemudian.
`\n`	Sesuai dengan *subexpression* ke-n dimana n bernilai 1 sampai 9.
`*`	Sesuai dengan elemen sebelumnya sebanyak 0 atau beberapa kali. Misal `ab*c` sesuai dengan "*ac*", "*abc*", "*abbbc*", dst. `[xyz]*` sesuai dengan "", "*x*", "*y*", "*z*", "*zx*", "*zyx*", "*xyzzy*", dst. `\(ab\)*` sesuai dengan "", "*ab*", "*abab*", "*ababab*", dst.
BRE: `\{m,n\}` ERE: `{m,n}`	Sesuai dengan elemen sebelumnya minimal sebanyak m kali namun tidak lebih dari n kali.
ERE: `?`	Sesuai dengan elemen sebelumnya sebanyak 0 atau 1 kali.
ERE: `+`	Sesuai dengan elemen sebelumnya minimal sebanyak 1 kali.
ERE: `|`	Operator alternasi yang sesuai dengan *expression* sebelumnya atau sesudahnya.

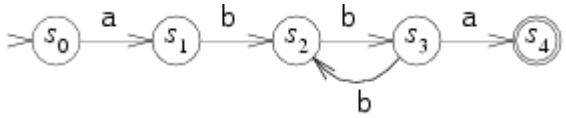
Misal:

- `.at` sesuai dengan tiga karakter apa saja yang diakhiri dengan "at", seperti "*hat*", "*cat*", dan "*bat*".
- `[hc]at` sesuai dengan "*hat*" dan "*cat*".
- `[^b]at` sesuai dengan semua string yang sesuai dengan `.at` kecuali "*bat*".
- `^[hc]at` sesuai dengan "*hat*" dan "*cat*", tetapi hanya yang ada pada awal string atau baris.
- `[hc]at$` sesuai dengan "*hat*" dan "*cat*", tetapi hanya yang ada pada akhir string atau baris.

2. 3. Finite Automata

Cara lain untuk mendeskripsikan himpunan karakter string adalah dengan *finite automata*. *Finite automata* juga dikenal sebagai *state machine*.

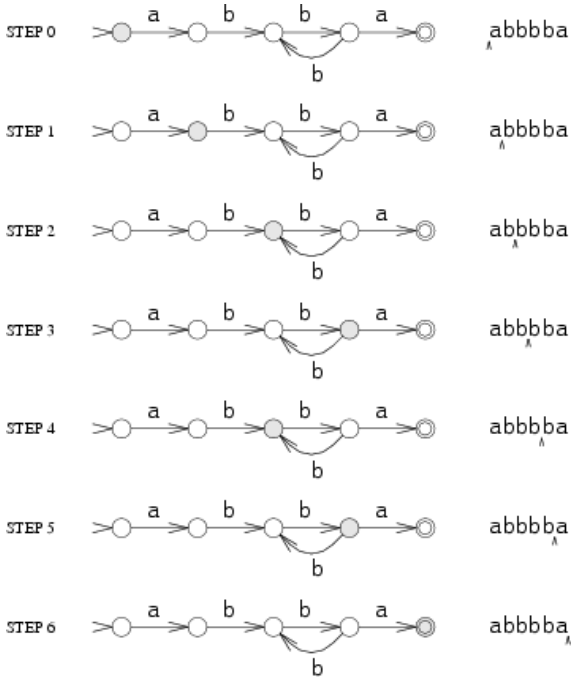
Sebagai contoh, mesin mengenal string yang sesuai dengan *regular expression* `a(bb)+a` sebagai:



Gambar 1 Contoh *finite automata*

Sebuah *finite automata* selalu berada pada salah satu *state* yang direpresentasikan oleh lingkaran pada diagram dengan label yang menunjukkan *state* yang berbeda dengan *state* lainnya. *State* awal digambarkan dengan panah masuk yang tidak berasal dari *state* lainnya, sementara *state* akhir digambarkan dengan lingkaran yang menutupi lingkaran *state* (dua lingkaran).

Berikut ilustrasi perpindahan *state* pada sebuah *finite automata*:



Gambar 2 Perpindahan *state* pada *finite automata*

3. IMPLEMENTASI

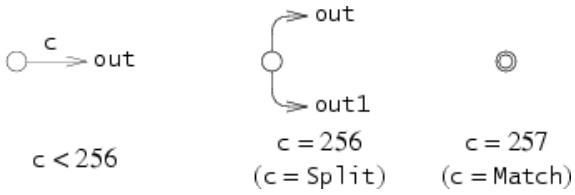
Thompson memperkenalkan pendekatan simulasi *multiple-state* pada tulisannya yang dimuat di Communication of the ACM pada tahun 1968. Pada bagian berikut akan disertakan implementasi algoritma pencarian string dengan *regular expression* dalam bahasa ANSI C. Implementasi pada bagian ini hanya terbatas pada ditemukan atau tidaknya sebuah pola pada teks. *Source code* lengkap tersedia *online* [3].

3. 1. Kompilasi ke NFA

Langkah pertama yang dilakukan adalah dengan mengkompilasi *regular expression* menjadi NFA yang sesuai. NFA akan digambarkan sebagai kumpulan *state* yang terhubung. Dimana *state* adalah:

```
struct State
{
        int c;
        State *out;
        State *out1;
        int lastlist;
};
```

Setiap `State` merepresentasikan salah satu dari tiga potongan NFA berikut bergantung pada nilai `c`, sementara `lastlist` akan dibahas pada bagian berikutnya.



Gambar 3 Tiga potongan NFA

Untuk menggambarkan *subexpression* perlu dibentuk sebuah struktur data yang menyimpannya, dalam hal ini disebut `Frag`.

```
Struct Frag
{
        State *start;
        Ptrlist *out;
};
Ptrlist *list1(State **outp);
Ptrlist *append(Ptrlist *l1,
Ptrlist *l2);
void patch(Ptrlist *l, State *s);
```

Penampung `start` menunjuk pada awal state pada *subexpression*, sementara `out` merupakan *list of pointers* yang belum terhubung pada apapun.

`List1` membuat *pointer list* baru dari *pointer* `outp`. `Append` menggabungkan dua *pointer lists*, kemudian mengembalikan hasilnya. `Patch` menghubungkan panah dari `l` ke `s`: hal ini akan melakukan *assignment* `*outp = s` untuk setiap *pointer* `outp` pada `l`.

Berikut fungsi utama dalam melakukan kompilasi ke NFA

```
State*
post2nfa(char *re)
{
        char *p;
```

```
        Frag stack[1000], *stackp, e1, e2,
e;
        State *s;

        #define push(s) *stackp++ = s
        #define pop()   *--stackp

        stackp = stack;
        for(p=re; *p; p++){
                switch(*p){
default:
        s = state(*p, NULL, NULL);
        push(frag(s, list1(&s->out));
        break;
case '|':
        e2 = pop();
        e1 = pop();
        s = state(Split, e1.start,
e2.start);
        push(frag(s, append(e1.out,
e2.out)));
        break;
case '?':
        e = pop();
        s = state(Split, e.start, NULL);
        push(frag(s, append(e.out,
list1(&s->out1))));
        break;
case '*':
        e = pop();
        s = state(Split, e.start, NULL);
        patch(e.out, s);
        push(frag(s, list1(&s->out1)));
        break;
case '+':
        e = pop();
        s = state(Split, e.start, NULL);
        patch(e.out, s);
        push(frag(e.start, list1(&s-
>out1)));
        break;
                }
        }

        e = pop();
        patch(e.out, matchstate);
        return e.start;
}
```

3. 2. Simulasi NFA

NFA disimpan pada sebuah `List` yang merupakan *array of* `State`.

```
struct List
{
        State **s;
        int n;
};
```

Simulasi ini menggunakan dua `list` yang dialokasikan secara global: `clist` merupakan himpunan *states* dimana NFA berada, dan `nlist` merupakan himpunan *states* berikutnya dimana NFA nantinya akan berada di sana, setelah memproses karakter saat itu.

```
int
match(State *start, char *s)
{
        List *clist, *nlist, *t;

        /* l1 and l2 are preallocated
globals */
        clist = startlist(start, &l1);
        nlist = &l2;
        for(; *s; s++){
                step(clist, *s, nlist);
                t = clist; clist =
nlist; nlist = t;  /* swap clist, nlist */
        }
        return ismatch(clist);
}
```

Apabila `List` akhir memiliki *matching state*, berarti string tersebut sesuai.

```
int
ismatch(List *l)
{
        int i;

        for(i=0; i<l->n; i++)
                if(l->s[i] ==
matchstate)
                        return 1;
        return 0;
}
```

Fungsi `addstate` akan menambahkan `State` pada `List` apabila belum terdapat pada `List` tersebut.

```
void
addstate(List *l, State *s)
{
        if(s == NULL || s->lastlist ==
listid)
                return;
        s->lastlist = listid;
        if(s->c == Split){
                /* follow unlabeled
arrows */
                addstate(l, s->out);
                addstate(l, s->out1);
                return;
        }
        l->s[l->n++] = s;
}
```

Fungsi `Startlist` membuat *initial state* pada `List` dengan cara hanya menambahkan `State` awal.

```
List*
startlist(State *s, List *l)
{
        listid++;
        l->n = 0;
        addstate(l, s);
        return l;
}
```

Fungsi `step` melanjutkan NFA menggunakan `clist` untuk melakukan komputasi terhadap `nlist` dan memproses karakter berikutnya.

```
void
step(List *clist, int c, List *nlist)
{
        int i;
        State *s;
```

```
          listid++;
          nlist->n = 0;
          for(i=0; i<clist->n; i++){
                    s = clist->s[i];
                    if(s->c == c)
                              addstate(nlist,
s->out);
          }
}
```

4. KESIMPULAN

Di antara berbagai algoritma pencarian string, penggunaan *regular expression* pada algoritma pencarian string dapat dilakukan dengan membangun NFA yang ekivalen terhadap *regular expression* yang menjadi pola pencarian terlebih dahulu, kemudian dilanjutkan dengan menyesuaikan teks dengan pola yang telah diterjemahkan menjadi NFA.

Penggunaan *regular expression* dimaksudkan untuk memperluas cakupan pencarian, mempersingkat pola yang dimasukkan sesuai dengan aturan yang ditentukan.

REFERENSI

[1] Thompson, Ken, "Regular Expression Search Algorithm", Communication of the ACM, Volume 11, Number 6, 1968, hal 419-422.

[2] http://www.ddj.com

[3] http://switch.com/~rsc/regexp

[4] http://www.wikipedia.org