Ekuivalensi Algoritma *Knuth-Morris-Pratt* dan *Finite State Automata*

Muhammad Atpur Rafif – 13522086[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
[1]13522086@std.stei.itb.ac.id

***Abstract*—Algoritma pencarian *string* memiliki banyak macamnya, seperti *Knuth-Morris-Prat* dan *Finite State Machine*. Penjelasan yang diberikan mengenai algoritma tersebut biasanya tidak mengaitkan hubungan antara algoritma tersebut. Makalah ini mengeksplorasi keterkaitan dan kesamaan dari kedua algoritma tersebut. Selain itu juga dilakukan eksperimen mengenai algoritma yang paling cepat berdasarkan pembahasan yang dilakukan.**

Keywords—string-matching; knuth-morris-pratt; finite-state-machine; equivalence;

I. Pendahuluan

Pencarian sebuah *string*, atau biasa disebut dengan *string matching* merupakan sebuah persoalan diberikan sebuah pola (*needle*) dan teks (*haystack*). Solusi dari persoalan ini adalah indeks dari pola pada teks tersebut. Besar dari teks relatif lebih besar dibandingkan dengan pola. Terdapat banyak kegunaan dari pencarian *string* ini, misalkan untuk teks editor. Penulis dapat mencari bagian yang ingin dihapus atau dirubah dengan menggunakan fitur pencarian.

Terdapat beberapa algoritma untuk menyelesaikan permasalahan ini. Salah satunya adalah algoritma *Knuth-Morris-Prat* ataupun *Finite State Machine*. Pada makalah ini akan dibahas mengenai keterkaitan dari kedua algoritma tersebut.

II. Landasan Teori

A. Pattern Matching

Sebuah permasalahan pencarian sebuah pola tertentu dari teks disebut dengan *pattern matching*. Panjang dari teks relatif jauh lebih besar dibandingkan dengan pola yang dicari. Pola tidak diharuskan memiliki bentuk pasti, namun bentuk dari pola yang dicari bisa bermacam-macam, misalkan pola yang diberikan memiliki bentuk sebuah kata yang diawali dengan huruf kapital. Sintaks yang biasa digunakan untuk merepresentasikan pola adalah *Regular Expresion* atau RegEx.

B. String Matching

Pada makalah ini hanya berfokus kepada permasalahan *pattern matching* yang spesifik, yaitu *string matching*. Kumpulan dari karakter atau huruf disebut dengan *string*. Pada permasalahan ini, bentuk dari pola adalah pasti. Pola yang diitemukan harus sama persis dengan yang dicari. Dengan kata lain hanya terdapat satu *string* yang memenuhi sebuah pola. Contoh dari pola tersebut adalah pola "needle", hanya terdapat satu *string* yangitu "needle" yang dapat memenuhi pola tersebut. Berbeda dengan pola "[A-Z][a-z]*" yang dapat memenuhi banyak *string* seperti "Needle" dan "Haystack". Indeks *string* pada makalah ini selalu dimulai dengan 0, untuk mempermudah implementasi.

C. Lazy Evaluation

Sebuah ekspresi dalam program dinyatakan sebagai *lazy* atau *non-strict* apabila nilai dari ekspresi tersebut tidak dihitung secara langsung, namun komputasi hanya dilakukan ketika nilai tersebut benar-benar dibutuhkan. Lawan dari konsep ini adalah *strict*, ketika ekspresi langsung dilakukan. Biasanya hal ini dilakukan untuk menghemat komputasi pada tahap berat dengan memajukannya diawal. Contoh dari metode *strict* ini adalah *lookup table*. Seluruh nilai yang dibutuhkan dihitung diawal.

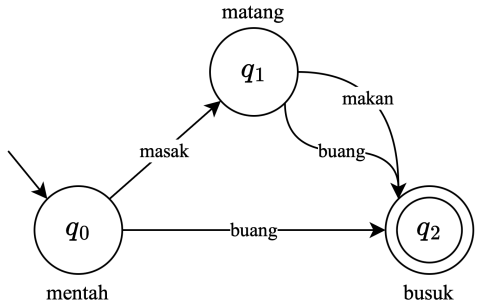
D. Finite State Machine

Sebuah mesin abstrak yang didefinisikan dari kumpulan *state* dan *transition function* disebut dengan *finite state machine*, atau disingkat dengan FSM. Sesuai dengan namanya, banyaknya *state* adalah terbatas. Kemudian *transition function* mengatur perpindahan dari *state* berdasarkan masukan atau *input* yang diberikan. Masukan ini biasa disebut dengan *alphabet*.

Mesin tersebut diberikan berbagai input dimulai dari *start state*. Setiap input yang diberikan akan mengubah *state* berdasarkan *transition function*. Kumpulan *input* yang diberikan disebut dengan *string*. Terdapat dua kemungkinan yang terjadi setelah sebuah FSM diberikan sebuah *input string*. Pertama ketika *state* berakhir pada *final state*. Kedua selain kejadian pertama. Sebuah *string* yang berakhir pada *final state* dikatakan *accepted* oleh mesin tersebut, selain itu disebut dengan *rejected*.

Sebuah analogi dari sebuah *state* adalah keadaan dari sebuah bahan makanan. Sedangkan *transition function* pada kasus ini adalah perlakuan kita terhadap bahan makanan tersebut. Beberapa *state* yang mungkin dari contoh ini adalah "mentah", "masak", dan "busuk". Sedangkan *transition* yang mungkin adalah dari "mentah" ke "busuk" setelah diberi perlakukan masak ke bahan tersebut.

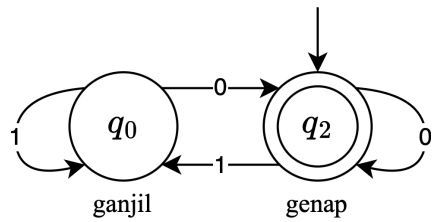
Sebuah bahan pasti dimulai dengan *state* "mentah" (*start state*) dan berakhir pada "busuk" (*final state*), baik oleh umur, atapun sistem pencernaan manusia. Banyak *Start state* selalu satu, sedangkan *final state* setidaknya satu, dan bisa lebih dari itu. Seluruh kumpulan perlakukan yang mungkin pada sebuah bahan tersebut akan diterima oleh mesin tersebut, sedangkan jika tidak maka akan ditolak. Berikut merupakan salah satu cara penulisan mesin tersebut:



Pada representasi diatas, *start state* dilambangkan dengan *state* yang memiliki panah tanpa asal. Sedangkan *final state* dilambangkan dengan dua lingkaran. Mesin diatas akan menerima "masak → makan" karena akan membuat mesin berakhir pada *final state*. Sedangkan *string* "buang → makan" akan ditolak, karena setelah masuk ke *state* "busuk" setelah "buang", tidak ada *transition function* dari "busuk" dengan masukan "makan". Tidak adanya *transition function* menyatakan bahwa mesin mati dan menolak *string* tersebut.

Permasalahan mengenai sebuah *string* diterima atau tidak disebut dengan *language* atau bahasa. Permasalahan dapat ditranslasikan menjadi bahasa. Misalkan penentuan sebuah bilangan prima. Sebuah representasi biner bilangan prima sebagai *string* akan diterima sebuah bahasa, selain itu akan ditolak. Jika terdapat sebuah mesin yang dapat menentukan sebuah *string* terdapat pada bahasa tersebut, maka kita bisa menentukan bilangan prima dengan *string* berupa representasi biner prima ke mesin tersebut dan mendapatkan hasil diterima atau ditolak. Terdapat bahasa yang dapat diselesaikan oleh FSM, namun terdapat juga bahasa yang membutuhkan konstruk lebih kompleks dibandingkan FSM untuk menyelesaikannya. Contoh konstruk tersebut adalah *push down automata* (disingkat PDA) atau *turing machine*.

Salah satu contoh bahasa yang dapat diselesaikan oleh FSM adalah menentukan bilangan ganjil atau genap. Input bilangan adalah representasi biner dari bilangan tersebut. Ketika dijalankan, *string* akan berakhir pada *final state* jika dan hanya jika bilangan tersebut adalah genap. Berikut merupakan gambaran representasinya:



Terdapat setidaknya empat representasi dari FSM ini [1]. Pada makalah ini hanya dibahas dua yang relevan. Berikut merupakan representasi dari FSM ini:

1) Deterministic Finite Automata (DFA)

Sebuah DFA dapat direpresentasikan sebagai sebuah *tuple* sebagai berikut:

$$A = (Q, \Sigma, \delta, q_0, F)$$

Penjelasan lebih lengkap sebagai berikut:

a) Q adalah himpunan dari *state* yang mungkin

b) Σ adalah himpunan dari *input* yang mungkin

c) δ adalah *transition function*

d) q_0 adalah *start state*

e) F adalah himpunan dari *final state*

Seluruh bagian sudah cukup jelas, kecuali δ. Fungsi tersebut menerima dua parameter sebagai *domain*, yaitu *state* dan *input*. Hasil dari fungsi atau *range* adalah *state*. Hal ini menentukan perpindahan dari *state* ketika diberikan *string*. Singkatnya $\delta(q_c, a) = q_n$, dengan q_c dan q_n adalah *state*, serta a adalah *alphabet*.

2) Non-Deterministic Finite Automata (NFA)

Sebuah NFA sama seperti dengan DFA, termasuk *tuple* representasinya. Namun terdapat satu perbedaan pada *transition function*. Parameter dari fungsi ini adalah sama, namun hasil dari fungsi ini adalah himpunan dari *state*. Ketika mesin ini dijalankan, mesin ini seakan "menerka" *state* saat ini. Hal ini seperti percabangan yang mungkin terjadi ketika mesin berjalan, atau terdapat beberapa *instance* dari mesin tersebut. Bentuk representasi ini sangat berguna untuk membuat algoritma *string matching* (Konteks *string matching* disini berbeda dengan *string* di FSM).

Kedua bentuk representasi diatas sama-sama dapat menerima bahasa yang disebut dengan *regular langguage*, sehingga kedua representasi tersebut *equivalence*. Sehingga kita bisa merubah representasi dari satu ke yang lainnya. Translasi NFA ke DFA mudah dilakukan, dengan mengubah δ_{DFA} memberikan keluaran himpunan dengan elemen hanya satu, yaitu keluaran dari fungsi DFA tersebut. Hal ini ditulis sebagai berikut:

$$\delta_{NFA}(q_c, a) = \{\delta_{DFA}(q_c, a)\}$$

Kemudian untuk seluruh bagian *tuple* yang lain adalah sama. Sedangkan translasi dari NFA ke DFA cukup kompleks. Prosedur untuk melakukan hal ini disebut dengan *subset construction*. Secara intuitif, kita bisa menjadikan seluruh terkaan sebuah NFA pada sebuah saat menjadi satu *state* tersendiri. Kemudian perpindahan *state* didasarkan pada terkaan pada saat ini. Secara lebih lengkap, berikut merupakan prosedur untuk melakukan *subset construction*:

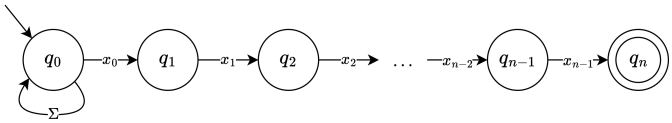
1) Sebuah *state* pada DFA hasil dituliskan sebagai himpunan *state* di NFA asal. Agar mempermudah penamaan, seluruh *state* pada himpunan akan disebut dengan elemen. Sedangkan himpunan disebut dengan *state*.

2) Berdasarkan poin 1, *start state* harus diubah untuk mengikuti aturan tersebut. Hasil *start state* yang baru pada DFA adalah $\{q_0\}$. Perhatikan disini bahwa pada DFA ini hanyalah satu *state*, sedangkan pada NFA ini merupakan himpunan dari *state*.
3) Dimulai dari *start state*, membuat *transition function* dengan cara sebagai berikut (Ingat bahwa di DFA, $\{q_{c_1}, \cdots, q_{c_n}\}$ merupakan satu *state*):

$$\delta_{DFA}(\{q_{c_1}, \cdots, q_{c_k}\}, a) = \bigcup_{i=1}^{k} \delta_{NFA}(q_{c_i}, a)$$

4) Pada setiap langkah pembuatan *transition function*, akan terdapat sebuah *state* DFA yang belum terdefinisi fungsinya. Buatlah *transition function* tidak ada lagi *transition* function yang belum terdefinisi sebelumnya.
5) Seluruh *state* DFA yang memiliki *final state* didalamnya menjadi *final state* yang baru.

Sebuah set yang memiliki banyak elemen n, memiliki 2^n subset. Sehingga translasi dari NFA ke DFA memiliki skenario terburuk eksponensial. Pencarian *string matching* dapat dilakukan dengan membuat NFA, melakukan translasi ke DFA, dan mensimulasikannya. Sebuah NFA untuk mencari sebuah *string* w yang berupa $x_0x_1 \cdots x_{n-1}$, dengan x_i adalah sebuah karakter adalah sebagai berikut:



Pada setiap masukan *alphabet*, mesin akan menerka antara awal pola sedang dimulai atau pola belum dimulai. Pada saat melakukan translasi dari NFA ke DFA dijamin bahwa DFA hasil pasti akan memiliki tepat $n + 1$ buah *state*.

E. Knuth-Morris-Pratt Algorithm

Pada permasalahan *string matching* terdapat beberapa algoritma yang dapat digunakan. Algoritma yang paling simpel adalah cara naif menggunakan *Brute Force*. Pada indeks dari teks akan diperiksa sudah menyerupai pola atau belum. Jika tidak maka indeks teks akan berpindah satu dan melakukan pengecekan yang sama.

Seiring berjalannya waktu, dibuatlah algoritma yang lebih sangkil dan mangkus yang dapat menyelesaikan permasalahan ini, seperti algoritma *Booyer-Moore* (BM) dan *Knuth-Morris-Pratt* (KMP). Pada makalah ini akan berfokus kepada algoritma kedua, yaitu KMP [2]. Algoritma ini dimulai dengan membuat *lookup table* berupa $\pi(i)$ untuk $i = 0 \cdots (n-1)$ dari w yang berupa $x_0x_1 \cdots x_{n-1}$. Nilai dari $\pi(i)$ adalah panjang *proper suffix* (bagian akhir *string*, atau *suffix* yang bukan berupa *string* itu sendiri) terbesar yang sama dengan *prefix* (bagian awal dari *string*) dari $x_0x_1 \cdots x_i$. Pastinya nilai dari $\pi(0)$ adalah 0 karena *proper suffix* yang ada hanyalah *string* kosong. Misalkan diberikan pola "abcaba", maka nilai dari $\pi(3)$ adalah 2, dikarenakan $x_0x_1 \cdots x_4$ = "abab" memiliki "ab" sebagai *proper suffix* terpanjang yang berupa *prefix* juga. Berikut merupakan nilai dari *lookup table* yang dibuat:

i	0	1	2	3	5	6
x_i	a	b	c	a	b	a
$\pi(i)$	0	0	0	1	2	1

Kemudian mencari pola pada teks dengan menggunakan bantuan *lookup table* yang telah dibuat. Sebuah variabel m yang menyatakan banyaknya karakter pada teks yang sama dengan *prefix* pada pola. Terdapat dua kemungkinan setiap iterasi indeks pada teks.

1) Pertama ketika pada indeks teks (misalkan i) sama dengan karakter pada indeks pola (disini juga menggunakan m sebagai indeks, namun bukan banyak karakter), maka kita bisa menambahkan kedua indeks dengan satu dan melanjutkan pencarian.
2) Kedua ketika karakter berbeda, maka kita tahu bahwa sudah terdapat m karakter yang sama dengan pola, dan ketika mencoba mecocokan di indeks m akan berbeda. Berdasarkan π, terdapat juga *prefix* di pola yang sudah cocok sebanyak $\pi(m)$. Maka kita bisa menggeser banyaknya karakter yang sudah cocok dengan $m \leftarrow \pi(m)$. Apabila setelah menggeser ternyata masih belum terdapat kecocokan pada m, maka dilakukan hal yang sama hingga $m = 0$. Hal ini berarti pada indeks i di teks tidak memiliki kesamaan pada *prefix* pola.

Algoritma berhenti ketika m adalah panjang dari pola. Apabila diterapkan menghasilkan langkah sebagai berikut:

Variabel			Teks											
i	m	tipe	a	b	a	a	b	c	a	b	c	a	b	a
0	0	1	**a**	b	c	a	b	a						
1	1	1	a	**b**	c	a	b	a						
2	2	2	a	b	**c**	a	b	a						
2	0	1			**a**	b	c	a	b	a				
3	1	2			a	**b**	c	a	b	a				
3	0	1				**a**	b	c	a	b	a			
4	1	1				a	**b**	c	a	b	a			
5	2	1				a	b	**c**	a	b	a			
6	3	1				a	b	c	**a**	b	a			
7	4	1				a	b	c	a	**b**	a			
8	5	2				a	b	c	a	b	**a**			
8	2	1							a	b	**c**	a	b	a
9	3	1							a	b	c	**a**	b	a
10	4	1							a	b	c	a	**b**	a
11	5	1							a	b	c	a	b	**a**
12	6	1	Berhenti, karena $m = 6$											

III. PEMBAHASAN

Buku Introduction to Automata Theory, Languages, and Computation hanya menyatakan bahwa translasi NFA ke DFA untuk *string matching* tidak akan mengubah banyaknya *state* tanpa memberikan bukti [3]. Sedangkan buku Introduction to Algorithm hanya menjelaskan kegunaan FSM untuk *string matching* tanpa mendalami konsep NFA dan DFA. Pada pembahasan ini, akan dibuktikan pernyataan translasi NFA ke DFA dan kegunaannya pada *string matching* menggunakan konsep *automata*.

Pada bagian ini akan dibahas mengenai ekuivalensi antara FSM dengan KMP. Sebelumnya perlu dibahas terlebih dahulu beberapa properti dari FSM untuk melakukan *string matching*. Pembuktian banyak menggunakan induksi. Pola disini adalah w yang berupa $x_0x_1 \cdots x_{n-1}$. Algoritma FSM dimulai dengan pembuatan NFA pada bab sebelumnya. Pada pembuatannya, *transition function* yang mungkin dari sebuah *state* q_i kecuali $i = 0$ adalah untuk menjadi q_{i+1} saat masukan adalah x_i. Selain itu maka *state* tersebut akan mati. Singkatnya untuk $i \neq 0$, $\delta_{NFA}(q_i, a) = \{\}$ untuk $a \in \Sigma$, dengan pengecualian $\delta_{NFA}(q_i, x_i) = \{q_{i+1}\}$. Dari sini dapat disimpulkan *state* q_{i+1} hanya dapat dihaslkan dari *transition function* dengan parameter q_i dan x_i.

Pada pembuatan NFA untuk *string matching*, sebuah *state* q_i memiliki arti bahwa prefix *string* masukan (teks) pada titik tersebut memiliki kesamaan dengan *prefix* pola dengan panjang i. Misalkan ketika mesin sudah mencapai q_2, hal ini berarti bentuk dari teks adalah $\cdots x_0x_1$. Terdapat dua kecocokan pada *suffix* teks dengan *prefix* pola.

> Lemma 3.1
> Seluruh *state* pada DFA untuk *string matching* selalu mengandung q_0 didalamnya

Menggunakan induksi, pada *start state* berbentuk $\{q_0\}$. Lalu kasus rekurens, misalkan sebuah *state* berbentuk $\{q_0, \cdots\}$. Maka untuk seluruh *state* yang dihasilkan dari *transition function* (Prosedur *subset construction*, menghasilkan *state* baru hanya melalui *transition function*) mengandung q_0 didalamnya. Hal ini dikarenakan $q_0 \in \delta_{NFA}(q_0, a)$ untuk seluruh $a \in \Sigma$.

> Lemma 3.2
> Jika mesin DFA untuk *string matching* berada pada *state* yang berupa $\{q_{c_1}, \cdots, q_{c_k}\}$, maka terdapat kecocokan dengan panjang c_i $(i = 1 \cdots k)$ antara *prefix* dari pola dan *suffix* dari teks saat ini untuk seluruh q_{c_i}.

Lemma diatas dapat dibuktikan dengan induksi. Basis terjadi ketika berada di *start state*, yaitu $\{q_0\}$. Hal ini jelas benar karena *prefix* dari pola dan *suffix* dari teks yang memiliki panjang nol selalu *string* kosong, dengan kata lain selalu sama.

Kemudian kasus rekurens, misalkan sekarang *state* DFA berada di $\{q_{c_1}, q_{c_2}, \cdots, q_{c_k}\}$. Ketika mesin DFA dimasukan sebuah *alphabet* baru, misalkan a. Berdasarkan NFA yang dibuat, setiap elemen pada *state* DFA saat ini memiliki dua kemungkinan sebagai berikut:

1) Sebuah elemen q_{c_i} akan mati, dikarenakan $a \neq x_{c_i}$ dan $\delta_{NFA}(q_{c_i}, a) = \{\}$. Karena kematiannya, maka elemen hasil dari *transition function* akan berkurang satu.
2) Sebuah elemen berpindah satu, atau $q_{c_i} \rightarrow q_{c_i+1}$ karena $a = x_{c_i}$ dan $\delta_{NFA}(q_{c_i}, x_{c_i}) = \{q_{c_i+1}\}$. Sehingga pada *transition function* akan muncul sebuah elemen baru.

Tambahan, berdasarkan lemma 3.1, q_0 selalu terbentuk untuk semua *state* pada DFA. Pada kasus pertama, tidak ada yang harus diperiksa dikarenakan *state* mati dan elemen tersebut tidak akan ada pada *state* yang dihasilkan *transition function*. Pada kasus kedua, awalnya q_{c_i} memiliki arti teks masukan memiliki bentuk $\cdots x_0x_1 \cdots x_{c_i-1}$ (panjang *suffix* teks saat ini yang sama dengan *prefix* pola adalah c_i) berdasarkan lemma.

Sekarang membuktikan *state* yang dihasilkan oleh *transition function* memenuhi lemma tersebut. Setelah dimasukan $a = x_{c_i}$ (karena kasus 2) pada mesin, berarti teks saat ini memiliki bentuk $\cdots x_0x_1 \cdots x_{c_i-1}x_{c_i}$, dan ini memiliki panjang kecocokan sebesar $c_i + 1$. Hal ini sesuai dikarenakan terdapat elemen q_{c_i+1} pada *state* yang dihasilkan oleh *transition function*. Pada kasus khusus q_0 selalu benar karena *string* kosong merupakan *prefix* dan *suffix* dari seluruh *string*.

> Lemma 3.3
> Jika terdapat sebuah *state* pada DFA untuk *string matching* yang memiliki bentuk $\{q_{c_1}, \cdots, q_{c_k}\}$, dengan $c_1 < c_2 < \cdots < c_k$ dan $c_k > 0$, maka terdapat juga sebuah *state* pada DFA yang memiliki bentuk $\{q_{c_1}, \cdots, q_{c_{k-1}}\}$.

DFA diawali dengan *start state* berupa $\{q_0\}$, namun tidak memenuhi syarat lemma diatas. Kemudian *state* baru selanjutnya yang pasti *terbentuk* adalah $\{q_0, q_1\}$. Hal ini dijadikan basis induksi, karena $\{q_0\}$ juga merupakan *state* dari seluruh DFA.

Kasus rekurens, misalkan terdapat *state* $\{q_{c_1}, \cdots, q_{c_k}\}$ dan $\{q_{c_1}, \cdots, q_{c_{k-1}}\}$ pada DFA. Kemudian dilakukan manipulasi sebagai berikut:

$$\begin{aligned}\delta_{DFA}(\{q_{c_1}, \cdots, q_{c_k}\}, a) &= \bigcup_{i=1}^{k} \delta_{NFA}(q_{c_k}, a) \\ &= \bigcup_{i=1}^{k-1} \delta_{NFA}(q_{c_k}, a) \cup \delta_{NFA}(q_{c_k}, a) \\ &= \delta_{DFA}(\{q_{c_1}, \cdots, q_{c_{k-1}}\}, a) \cup \delta_{NFA}(q_{c_k}, a)\end{aligned}$$

Nilai *state* yang dihasilkan $\delta_{DFA}(\{q_{c_1}, \cdots, q_{c_{k-1}}\}, a)$ akan memiliki bentuk $\{q_{n_1}, \cdots, q_{n_l}\}$, dengan $n_l \leq c_k$. Menggunakan induksi kuat, maka $\{q_{n_1}, \cdots, q_{n_l}\}$ juga terdapat pada DFA. Pada persamaan hasil manipulasi diatas, *state* baru terbentuk ketika

$a = x_{c_k}$. Dengan begitu, *state* baru tersebut memiliki bentuk $\{q_{n_1}, \cdots, q_{n_l}\} \cup \{q_{c_k+1}\}$. Sehingga kasus rekurens terbukti.

Lemma 3.4
Definisi c_k merupakan nilai tertinggi pada *state* DFA untuk *string matching* berbentuk $\{q_{c_1}, \cdots, q_{c_k}\}$ jika $(c_k > c_{k-1} > \cdots > c_1)$.

Bagi seluruh $i = 0 \cdots n$, hanya terdapat tepat satu *state* dimana i merupakan nilai tertinggi *state* tersebut.

Basis pada *start state* berupa $\{q_0\}$ adalah trivial, karena tidak ada lagi *state* dengan nilai tertinggi 0 yang mungkin selain itu. Pada kasus induksi, misalkan nilai tertinggi c_k hanya dimiliki oleh $\{q_{c_1}, \cdots, q_{c_k}\}$. Elemen $q_{c_{k+1}}$ hanya bisa didapat dari elemen q_{c_k}. Misalkan hasil dari *transition function* adalah $\{q_{n_1}, \cdots, q_{n_l}\}$, maka berdasarkan lemma 3.3, $\{q_{n_1}, \cdots, q_{n_{l-1}}\}$ sudah merupakan bagian dari DFA, sehingga tidak terbentuk *state* baru. Sedangkan ketika $a = x_{c_k}$, maka akan terbentuk sebuah *state* baru berupa $\{q_{n_1}, \cdots, q_{n_l}\}$, dengan $n_l = c_k + 1$ yang memiliki $c_k + 1$ sebagai nilai tertinggi *state* DFA tersebut.

Kasus induksi terbukti hanya membuat tepat satu *state* DFA baru dengan nilai tertinggi $c_k + 1$. Hal ini terus terjadi hingga $c_k = n$ dan berhenti karena sudah tidak ada lagi *transition function* pada NFA awal. Menggunakan lemma ini, tidak perlu lagi menuliskan *state* pada DFA dengan himpunan $\{q_{c_1}, \cdots, q_{c_k}\}$, menulis q_{c_k} saja sudah cukup. Menggunakan teorema ini, kita bisa menuliskan hasil manipulasi dari persamaan pada lemma 3.3 menjadi berikut ini:

$$\delta_{DFA}(q_{c_k}, a) = \delta_{DFA}(q_{c_{k-1}}, a) \cup \delta_{NFA}(q_{c_k}, a)$$

Teorema 3.1
Translasi dari NFA ke DFA untuk *string matching* dengan panjang pola n tepat menghasilkan $n + 1$ *state*, berupa $q_0 \cdots q_n$ dengan interpretasi mesin akan memasuki *state* q_n setelah masukan *string* dari teks apabila *suffix* dari teks yang dimasukan sama dengan *prefix* dari pola.

Teorema diatas merupakan hasil langsung gabungan antara lemma 3.2 dan lemma 3.4. Sebuah *state* q_{c_k} pada DFA memiliki bentuk asli $\{q_{c_1}, \cdots, q_{c_k}\}$, ($c_i$ terurut dari terendah). Nilai i pada q_i memiliki arti panjang *prefix* dari pola berdasarkan lemma 3.2, serta $\cdots x_0 \cdots x_{c_k}$ memiliki *suffix* yang sama dengan $\cdots x_0 \cdots x_{c_{k-1}}$ yang keduanya memiliki kecocokan dengan pola. Maka nilai dari $x_{c_{k-1}}$ adalah sama seperti *lookup table* pada KMP yang berupa π. Ilustrasinya bisa digambarkan sebagai berikut:

q_{c_k}	=	$\cdots$	x_0	$\cdots$	$x_{c_{k-1}}$	$\cdots$
$q_{c_{k-1}}$	=	$\cdots$			x_0	

Sehingga kita bisa menyederhakan lagi persamaan pada lemma 3.4 menjadi:

$$\delta_{DFA}(q_{c_k}, a) = \delta_{DFA}(q_{\pi(c_k)}, a) \cup \delta_{NFA}(q_{c_k+1}, a)$$

Berdasarkan lemma 3.4 dan pembuktian lemma 3.2, fungsi diatas dapat disederhanakan lagi berdasarkan masukan *alphabet* dari a. Kemudian sebuah *state* q_s disederhanakan menjadi s.

$$\delta(s, a) = \begin{cases} \delta(\pi(s), a) \text{ if } a \neq x_s \\ 0 \text{ if } a \neq x_s \text{ and } s = 0 \\ s + 1 \text{ if } a = x_s \end{cases}$$

Mesin DFA akan berhenti dan memasuki *final state* ketika nilai i adalah panjang dari pola. Menggunakan fungsi tersebut, dapat dibuat langsung *lookup table* dari seluruh *state* dan *alphabet* yang ada. Namun fungsi tersebut dapat diimplementasikan dan disimulasikan secara langsung tanpa menggunakan *lookup table*. Menggunakan cara tersebut, menghasilkan *pseudocode* sebagai berikut:

```
// W pola, T teks
s ← 0
for i = 0 to m
   // Nilai m adalah panjang teks
   // W[s] adalah xs
   // T[i] adalah a

   // Rekursif δ dan kasus dua
   while s > 0 and W[s] ≠ T[i]
        s ← π[s]

   // Basis kasus satu
   if W[s] = T[i]
        s ← s + 1

   // Final state
   if s = n
        print "Pola ditemukan"
        break
```

Bentuk algoritma diatas mirip dengan algoritma pada KMP. Bagian rekursif dan kasus dua sama seperti kemungkinan kedua setiap iterasi dari algoritma KMP ketika indeks dari teks tidak sama dengan indeks dari pola. Sedangkan basis kasus satu sama dengan kemungkinan satu pada algoritma KMP, ketika karakter di indeks pola dan teks adalah sama. Nilai dari s pada FSM sama seperti nilai m pada KMP. Dari seluruh pembahasan yang telah dilakukan, diambil kesimpulan bahwa

Teorema 3.2
Algoritma KMP adalah algoritma FSM tanpa menggunakan *lookup table*. Sedangkan FSM adalah algoritma KMP menggunakan *lookup table*.

Secara teori, membentuk *lookup table* merupakan hal yang sepadan dengan waktu pembuatannya apabila panjang dari teks sangat besar, sehingga program tidak perlu melakukan perhitungan secara berulang. Bagian eksperimen akan menelusuri hal ini secara langsung.

IV. Eksperimen

Program dibuat menggunakan Bahasa C, dari bab pembahasan dibuat empat jenis algoritma. Dua algoritma pertama sama seperti penjelasan KMP dan FSM sebelumnya. Terdapat tambahan dua algoritma dengan penjelasan sebagai berikut:

1) fsm_compiled

   Penggunaan *lookup table* pada algoritma FSM memberikan sebuah *bottleneck* atau pembatas berupa kecepatan membaca *lookup table* dari memori. Hal tersebut dapat diatasi dengan memanfaatkan optimasi sintaks *case* pada Bahasa C [4]. Algoritma ini akan membuat *macro* dari definisi sebuah DFA.

2) combined

   Merupakan gabungan antara FSM dan KMP. Pada kemugkinan kedua pada setiap iterasi KMP terdapat kemungkinan melakukan iterasi lagi hingga didapat indeks pola sehingga karakter pola dan teks sama, atau nilai dari indeks pola adalah nol. Penggunaan *lookup table* membuat iterasi dalam tidak diperlukan. Nilai *lookup table* hanya digunakan ketika indeks pola tidak nol, karena frekuensi indeks pola nol relatif besar, serta nilai dari $\pi(0)$ dipastikan adalah nol. Memberikan sebuah kasus konstan lebih cepat dibandingkan dengan membaca memori untuk *lookup table*.

Seluruh algoritma diatas dijalankan menggunakan 16-inch Macbook Pro M1 Max, pada sebuah file dengan banyak karakter acak sebesar `INT_MAX` pada Bahasa C. Pola yang dicari adalah “abacabadabacabaae". Parameter eksekusi adalah waktu pencarian, tanpa menghitung waktu *load file* ke memori, serta banyaknya iterasi yang dilakukan (termasuk rangkap pada KMP). Berikut merupakan hasilnya:

Run	kmp	fsm	fsm_compiled	combined
Iterasi				
-	2155971253	2147483563		
Waktu (Detik)				
1	4.177	4.849	2.166	2.177
2	4.206	4.870	2.180	2.163
3	4.207	4.878	2.191	2.175
4	4.175	4.826	2.162	2.190
5	4.213	4.866	2.187	2.167
mean	4.196	4.858	2.177	2.174

Berdasarkan eksperimen yang dilakukan, algoritma “combined” memiliki waktu yang paling cepat, dekat dengan “fsm_compiled”. Namun pada “fsm_compiled”, waktu kompilasi dari *macro* tidak dihitung, sehingga algoritma “combined” masih lebih baik. Meskipun banyak selisih iterasi antara “fsm” dan “kmp” sebesar 8.487.690, namun algoritma “fsm” lebih lambat dari “kmp” karena *lookup table* terkena batasan kecepatan memori. Selisih ini jauh lebih besar dibandingkan iterasi yang diperlukan untuk membuat *lookup table* pada algoritma FSM, yaitu (`UCHAR_MAX`) * (Besar pola) = 4.352.

V. Kesimpulan

Berdasarkan pembahasan yang telah diberikan, *string matching* menggunakan algoritma FSM dan KMP adalah sama. Perbedaanya hanya terletak pada pembuatan *lookup table*. Kemudian dilakukan eksperimen dan didapat bahwa *lookup table* memiliki keterbatasan, yaitu kecepatan dari memori komputer. Cara terbaik yang dapat digunakan adalah gabungan dari dari FSM dan KMP, seperti yang dijelaskan pada algoritma “combined” pada bab eksperimen.

VI. Lampiran dan Catatan

Link *Source Code* dari eksperimen berada di https://github.com/atpur-rafif/IF2211. Seluruh gambar dibuat oleh penulis. Gambar yang berupa diagram dibuat menggunakan https://app.diagrams.net. Seluruh lemma dan teorema pada bagian pembahasan merupakan pemikiran dari penulis.

Ucapan Terima Kasih

Penulis berterimakasih kepada Allah S.W.T. dengan izin-Nya makalah ini selesai ditulis. Kemudian kepada orang tua penulis yang telah mendukung penulis sampai saat ini. Selanjutnya kepada seluruh dosen mata kuliah strategi algoritma sebagai inspirasi pembuatan makalah ini. Serta masih banyak pihak lainnya yang tidak bisa disebutkan satu-persatu.

Referensi

[1] John E. Hopcroft, “Introduction to Automata Theory, Languages, and Computation“, 3rd ed. Boston: Pearson, 2007, pp. 92-93.

[2] Thomas H. Cormen, “Introduction to Algorithm”, 4th ed. Cambridge: MIT Press, 2022, pp. 975-985

[3] John E. Hopcroft, “Introduction to Automata Theory, Languages, and Computation“, 3rd ed. Boston: Pearson, 2007, pp. 69-70

[4] Randall E. Bryant, “Computer System A Programmer’s Perspective”, 4th ed. Boston: Pearson, 2011, pp. 213-217

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 Juni 2024
Muhammad Atpur Rafif