TUGAS AKHIR – KI141502

IMPLEMENTASI MODEL DETERMINISTIC FINITE AUTOMATON UNTUK INTERPRETASI REGULAR EXPRESSION PADA STUDI KASUS PERMASALAHAN SPOJ KLASIK 10354

MUHAMMAD YUNUS BAHARI
NRP 5111100079

Dosen Pembimbing I
Arya Yudhi Wijaya, S.Kom., M.Kom.

Dosen Pembimbing II
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2015

TUGAS AKHIR – KI141502

IMPLEMENTASI MODEL DETERMINISTIC FINITE AUTOMATON UNTUK INTERPRETASI REGULAR EXPRESSION PADA STUDI KASUS PERMASALAHAN SPOJ KLASIK 10354

MUHAMMAD YUNUS BAHARI
NRP 5111100079

Dosen Pembimbing I
Arya Yudhi Wijaya, S.Kom., M.Kom.

Dosen Pembimbing II
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2015

[Halaman ini sengaja dikosongkan]

UNDERGRADUATE THESES – KI141502

IMPLEMENTATION OF DETERMINISTIC FINITE AUTOMATON MODEL FOR REGULAR EXPRESSION INTERPRETATION A CASE STUDY AT SPOJ CLASSIC PROBLEM 10354

MUHAMMAD YUNUS BAHARI
NRP 5111100079

Supervisor I
Arya Yudhi Wijaya, S.Kom., M.Kom.

Supervisor II
Rully Soelaiman, S.Kom., M.Kom.

DEPARTMENT OF INFORMATICS
FACULTY OF INFORMATION TECHNOLOGY
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA 2015

[Halaman ini sengaja dikosongkan]

LEMBAR PENGESAHAN

IMPLEMENTASI MODEL *DETERMINISTIC FINITE AUTOMATON* UNTUK INTERPRETASI *REGULAR EXPRESSION* PADA STUDI KASUS PERMASALAHAN SPOJ KLASIK 10354

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Desain dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh
MUHAMMAD YUNUS BAHARI
NRP : 5111 100 079

Disetujui oleh Dosen Pembimbing Tugas Akhir:

1. Arya Yudhi Wijaya, S.Kom., M.Kom. .....
   NIP: 198409042010121002 (Pembimbing 1)

2. Rully Soelaiman, S.Kom., M.Kom. .....
   NIP: 197002131994021001 (Pembimbing 2)

SURABAYA
MEI, 2015

[Halaman ini sengaja dikosongkan]

Implementasi Model *Deterministic Finite Automaton* untuk Interpretasi *Regular Expression* pada Studi Kasus Permasalahan SPOJ Klasik 10354

Nama Mahasiswa	**: MUHAMMAD YUNUS BAHARI**
NRP	**: 5111100079**
Jurusan	**: Teknik Informatika FTIF-ITS**
Dosen Pembimbing 1	**: Arya Yudhi Wijaya, S.Kom., M.Kom.**
Dosen Pembimbing 2	**: Rully Soelaiman, S.Kom., M.Kom.**

Abstrak

Regular expression merupakan salah satu bentuk pola yang banyak digunakan untuk melakukan pencarian dan validasi string. Namun pada implementasinya sering kali regular expression hanya ditransformasi menjadi sebuah model Nondeterministic Finite Automaton (NFA) yang masih memiliki kekurangan apabila digunakan untuk melakukan validasi dan pencarian string. Kekurangan tersebut adalah lamanya waktu yang diperlukan untuk melakukan proses pencarian dan validasi.

Pengembangan pencarian dan validasi dari model NFA terus berkembang dan memunculkan teknik konversi model NFA menjadi model Deterministic Finite Automaton (DFA) untuk menambah performa pencarian dan validasi. Dalam Tugas Akhir ini akan dibahas mengenai teknik yang digunakan dalam konversi model NFA ke DFA dan bagaimana implementasi tersebut dalam program.

Kata kunci: Deterministic Finite Automaton, Nondeterministic Finite Atomaton, Regular expression, SPOJ.

[Halaman ini sengaja dikosongkan]

IMPLEMENTATION OF DETERMINISTIC FINITE AUTOMATON MODEL FOR REGULAR EXPRESSION INTERPRETATION A CASE STUDY AT SPOJ CLASSIC PROBLEM 10354

Student's Name	**: MUHAMMAD YUNUS BAHARI**
Student's ID	**: 5111100079**
Department	**: Teknik Informatika FTIF-ITS**
First Advisor	**: Arya Yudhi Wijaya, S.Kom., M.Kom.**
Second Advisor	**: Rully Soelaiman, S.Kom., M.Kom.**

Abstract

Regular expression is a pattern that often used for searching and validate a string. But the implementation of regular expression often just transformed into Nondeterministic Finite Automaton (NFA) that still has flaws. One main flaw is NFA took a long time to do a validation and string matching.

The development of validation and string matching is continuously expanding and some of discoveries are made to improve validation and string matching performance. One of them is converting NFA into Deterministic Finite Automaton (DFA) to reduce active states in NFA. In this final project conversion technique from NFA to DFA and how to implement that technique into a program will be explained.

Keywords: Regular expression, Nondeterministic Finite Automaton, Deterministic Finite Automaton, SPOJ.

[Halaman ini sengaja dikosongkan]

KATA PENGANTAR

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

Puji syukur penulis kehadirat Tuhan YME karena berkat rahmat dan karunia-NYA penulis dapat menyelesaikan Tugas Akhir yang berjudul

IMPLEMENTASI MODEL *DETERMINISTIC FINITE AUTOMATON* UNTUK INTERPRETASI *REGULAR EXPRESSION* PADA STUDI KASUS PERMASALAHAN SPOJ KLASIK 10354

Tugas Akhir ini merupakan salah satu syarat untuk memperoleh gelar Sarjana Komputer di Jurusan Teknik Informatika, Fakultas Teknologi Informasi, Institut Teknologi Sepuluh Nopember Surabaya.

Penulis ingin menyampaikan terima kasih yang sebesar-besarnya atas dukungan dan semangat yang diberikan dan membantu penulis baik secara langsung ataupun tidak dalam menyelesaikan Tugas Akhir ini. Penulis ingin mengucapkan terima kasih kepada

1. Tuhan YME karena berkat rahmat dan karunianya penulis berhasil menyelesaikan Tugas Akhir dengan baik.
2. Kedua orang tua, dan keluarga penulis, terima kasih atas doa dan bantuan moral dan material selama penulis belajar di Teknik Informatika ITS.
3. Ibu Dr. Eng. Nanik Suciati, S.Kom., M.Kom., selaku ketua jurusan Teknik Informatika ITS
4. Bapak Radityo Anggoro, S.Kom., M.Sc. selaku Koordinator Tugas Akhir di Teknik Informatika ITS.
5. Bapak Arya Yudhi W., S.Kom., M.Kom. selaku Dosen Pembimbing I Tugas Akhir yang telah memberikan bimbingan dan dukungan selama penulis menyelesaikan Tugas Akhir.

6. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku pembimbing II Tugas Akhir yang telah memberikan banyak waktu untuk berdiskusi dan memberi semangat dan motivasi kepada penulis untuk menyelesaikan Tugas Akhir.
7. Bapak dan Ibu Dosen di Jurusan Teknik Informatika yang telah memberikan ilmu selama penulis kuliah di Teknik Informatika
8. Seluruh Staf dan karyawan Teknik Informatika yang telah memberikan bantuan selama penulis kuliah di Teknik Informatika.
9. Rekan-rekan kader terbaik bangsa, Duddin, Indra, Ivan dan Sindu yang telah bersedia memberikan ilmu yang bermanfaat dan bantuan selama penulis belajar di Teknik Informatika ITS.
10. Rekan-rekan TC Hura-hura yang telah setia menjadi teman traveling penulis dan telah memberikan dukungan pada penulis.
11. Rekan-rekan di laboratorium Algoritma dan Pemrograman yang telah bersedia dan betah dengan adanya penulis di lab selama pengerjaan Tugas Akhir.
12. Rekan-rekan lain yang tidak bisa disebutkan satu-persatu di sini, TC2011, theContracant, rekan pemain DOTA TC dan seluruh pihak yang telah bersedia memberikan hiburan, semangat dan motivasi ke penulis.

Penulis Mohon maaf apabila terdapat kekurangan dalam penulisan Tugas Akhir ini. Kritik dan saran penulis harapkan untuk perbaikan dan pembelajaran di kemudian hari. Semoga Tugas Akhir ini dapat memberikan Manfaat yang sebesar besarnya.

Surabaya, Mei 2015

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

[Halaman ini sengaja dikosongkan]

DAFTAR KODE SUMBER

[Halaman ini sengaja dikosongkan]

BAB I
PENDAHULUAN

1.1 Latar Belakang

Topik Tugas Akhir ini diangkat dari permasalahan yang terdapat pada *Online Judge* SPOJ dengan nomor soal 10354 dan kode CTSTRING. Pada permasalahan ini diberikan dua buah masukan yaitu sebuah *string* yang merupakan *Regular Expression* dan sebuah bilangan L. Dari dua masukan tersebut kita diminta untuk menentukan berapa banyak *string* dengan panjang L yang memenuhi *Regular Expression* yang diberikan.

Regular Expression merupakan metode yang umum digunakan dalam pemrosesan *string*. Sebagai contoh untuk pengecekan alamat email yang valid dan pencarian kata dalam dokumen. Untuk menyelesaikan permasalahan tersebut diperlukan cara untuk menginterpretasikan *Regular Expression* ke model yang dapat diterjemahkan ke dalam logika pemrograman. *Deterministic Finite Automaton* merupakan model dengan keadaan tertentu yang dapat berubah menjadi keadaan yang lain mengikuti masukan tertentu yang dapat diterjemahkan ke dalam logika pemrograman.

Oleh karena itu, dalam penyelesaian studi kasus SPOJ klasik 10354 teknik pemodelan yang digunakan untuk menginterpretasikan *Regular Expression* adalah *Deterministic Finite Automaton*.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam Tugas Akhir ini sebagai berikut,

1. Bagaimana memodelkan *regular expression* ke dalam bentuk *Deterministic Finite Automaton* ?
2. Bagaimana mengimplementasikan model *Deterministic Finite Automaton* untuk menginterpretasikan *regular expression*?

3. Bagaimana menguji kebenaran dan kinerja model yang telah diimplementasi ke dalam program?

1.3 Batasan Masalah

Permasalahan yang dibahas pada Tugas Akhir ini memiliki beberapa batasan, yaitu sebagai berikut,

1. Implementasi dilakukan dengan bahasa pemrograman C++.
2. Terdapat dua masukan program yakni *regular expression* dan sebuah bilangan L.
3. Batas maksimum jumlah karakter *regular expression* adalah 100 karakter.
4. Alfabet yang digunakan dalam *regular expression* ada dua yakni ‘a’ dan ‘b’.
5. Operasi yang dapat dilakukan dalam *regular expression* ada tiga yakni *concatenation* `(ab)`, *union* `(a|b)` dan *star quantifier/Klenee star* `(a*)`.
6. Batas maksimum Nilai L adalah 10^9.

1.4 Tujuan

Tujuan dari Tugas Akhir ini adalah sebagai berikut,

1. Mengetahui, memahami, dan mengimplementasikan interpretasi *regular expression* kedalam bentuk *Deterministic Finite Automaton*.
2. Mengetahui, dan memahami pengaplikasian model *Deterministic Finite Automaton* untuk interpretasi *regular expression*.
3. Mengetahui dan menguji kebenaran dan kinerja program yang diimplementasikan.

1.5 Manfaat

Tugas Akhir ini diharapkan dapat membantu memahami penggunaan *Deterministic Finite Automaton* sebagai salah satu teknik pemodelan untuk menginterpretasikan *regular expression*.

1.6 Metodologi

Tahapan-tahapan yang dilakukan dalam pengerjaan Tugas Akhir ini adalah sebagai berikut,

1. Penyusunan proposal Tugas Akhir.

   Tahap awal untuk memulai pengerjaan Tugas Akhir ada penyusunan proposal Tugas Akhir. Pada proposal ini, penulis mengajukan Implementasi Model *Deterministic Finite Automaton* untuk Interpretasi *regular expression* pada Studi Kasus Permasalahan SPOJ Klasik 10354.

2. Studi literatur

   Pada tahap studi literatur penulis melakukan pencarian informasi yang diperlukan untuk penyelesaian permasalahan yang akan dikerjakan. Informasi untuk penyelesaian permasalahan didapatkan dari buku acuan yang berhubungan dengan algoritma penyelesaian permasalahan yang diangkat pada Tugas Akhir ini.

3. Implementasi perangkat lunak

   Tahap implementasi merupakan tahap untuk mengubah rancangan algoritma yang diperoleh dan disusun dari berbagai literatur yang kemudian disesuaikan dengan permasalahan terkait. Implementasi program ini ditulis dengan menggunakan bahasa pemrograman C++.

4. Pengujian dan evaluasi

   Pada tahap ini penulis melakukan uji coba dengan melakukan pengiriman kode program pada *Online Judge* SPOJ sesuai dengan permasalahan yang terkait apakah solusi yang diusulkan sudah sesuai dengan kriteria permasalahan yang ada.

5. Penyusunan buku Tugas Akhir.

   Pada tahap ini dilakukan penyusunan laporan yang menjelaskan dasar teori dan metode yang digunakan dalam tugas akhir ini serta hasil dari implementasi program yang telah dibuat.

[Halaman ini sengaja dikosongkan]

BAB II
TINJAUAN PUSTAKA

Bab ini berisi penjabaran teori yang berkaitan dengan algoritma yang digunakan dalam implementasi program. Penjelasan dalam bab ini berguna untuk memberikan gambaran bagaimana program yang dibuat bekerja dan diharapkan dengan adanya penjelasan berikut pengembangan metode yang diimplementasikan akan lebih mudah.

2.1 Regular Expression

Sebuah *string* didefinisikan sebagai rangkaian karakter berhingga yang ada dalam semesta simbol Σ. Sebagai contoh FAZZIBASHASOL adalah *string* dalam semesta simbol $\Sigma = \{A, B, F, H, I, L, O, S, Z\}$ [1]. *Regular expression* adalah salah satu cara yang sering digunakan untuk merepresentasikan pola pencarian yang lebih kompleks dibandingkan dengan sebuah *string* ataupun sekumpulan *string*. *Regular expression* didefinisikan sebagai berikut.

regular expression RE adalah sebuah *string* yang memiliki simbol dalam set $\Sigma \cup \{\varepsilon, |, *, \cdot, (,)\}$, yang didefinisikan secara rekursif sebagai *string* kosong ε, simbol α di mana $\alpha \in \Sigma$, dan (RE_1), $(RE_1 *)$, $(RE_1|RE_2)$, dan $(RE_1 \cdot RE_2)$, di mana RE_1 dan RE_2 adalah *regular expression* [1].

Salah satu contoh *regular expression* adalah $(((A \cdot B) *)|(B \cdot B))$. Untuk kemudahan membaca *regular expression* sering kali $(RE_1 \cdot RE_2)$ ditulis menjadi $(RE_1 RE_2)$. Sehingga contoh *regular expression* tersebut dapat disederhanakan menjadi $(((AB) *)|(BB))$. Simbol "A","B" adalah alfabet dari *regular expression* sedangkan "*", "|", "•" adalah operator *regular expression*.

Sebuah bahasa (Kumpulan dari *string) L* yang direpresentasikan oleh *regular expression RE* adalah bahasa

dengan simbol α dimana $\alpha \epsilon \Sigma$ yang memiliki struktur *RE* sebagai berikut [1],

- Jika RE adalah ε, maka $L(RE) = \{\varepsilon\}$, *string* kosong.
- Jika RE adalah $\alpha \in \Sigma$, maka $L(RE) = \{\alpha\}$, sebuah *string* dengan panjang satu karakter.
- Jika RE adalah *regular expression* dengan bentuk RE_1, maka $L(RE) = L(RE_1)$.
- Jika RE adalah *regular expression* dengan bentuk $(RE_1 \cdot RE_2)$, maka $L(RE) = L(RE_1) \cdot L(RE_2)$. di mana $W_1 \cdot W_2$ adalah kumpulan string w sedemikian hingga $w = w_1 w_2$, dimana $w_1 \in W_1$ dan $w_2 \in W_2$. Operator "·" adalah operator *concatenate*.
- Jika RE adalah *regular expression* dengan bentuk $(RE_1 | RE_2)$, maka $L(RE) = L(RE_1) \cup L(RE_2)$. Operator "|" adalah operator *Union.*
- Jika RE adalah *regular expression* dengan bentuk $(RE_1 *)$, maka $L(RE) = L(RE_1) * = \bigcup_{i \geq 0} L(RE_1)^i$. di mana $L^0 = \{\varepsilon\}$ dan $L^i = L \cdot L^{i-1}$ untuk semua L. Operator "*" adalah operator *Klenee Star*.

Sebagai contoh *regular expression* $\Big(\big((AB) *\big) \Big| (BB)\Big)$ akan merepresentasikan bahasa $L\Big(\big((AB) *\big) \Big| (BB)\Big) = \{\varepsilon, AB, BB, ABAB, ABABAB, ABABAB, \dots\}$.

2.2 String Matching Menggunakan Regular Expression

Mencari *string* dalam sebuah kumpulan *string* dengan menggunakan *regular expression* diperlukan untuk mengubah bentuk *regular expression* menjadi sebuah *state machine* yang disebut *Nondeterministic Finite Automaton* (NFA). Ada dua metode yang umum digunakan dalam melakukan konversi *regular expression* menjadi NFA, yakni dengan menggunakan metode Thompson dan menggunakan metode Glushkov. Di buku ini penulis akan menjelaskan metode Thompson pada subbab 2.4.

Hasil konversi *regular expression* ke NFA akan menghasilkan sebuah *state machine* yang bisa digunakan untuk melakukan pencarian *string* dalam kumpulan *string*. Teknik menggunakan NFA sebagai pencari *string* yakni dengan memasukkan setiap simbol dalam *string* dari awal *string* hingga akhir *string* pada *state* awal sebuah NFA, atau hingga sebuah NFA tidak menemui masukan simbol yang valid untuk menuju *state* berikutnya dari simbol yang diberikan secara berurutan sehingga masukan simbol berikutnya akan kembali dimulai dari *state* awal NFA. Apabila suatu simbol dari *string* mencapai *state* selesai dari NFA maka sebuah *string* dinyatakan cocok dengan pola *regular expression* yang diberikan.

Dalam penggunaannya NFA memiliki kelemahan yakni waktu yang digunakan untuk melakukan pencarian lambat apabila terdapat banyak *state* dalam NFA. Untuk meningkatkan performa pencarian *string*, NFA dapat dikonversikan lagi menjadi *Deterministic Finite Automaton* (DFA) dengan menghilangkan *state* transisi kosong sehingga untuk setiap simbol yang dimasukkan hanya akan nada satu *state* yang aktif, konversi dari NFA ke DFA akan dibahas pada subbab 2.6.

2.3 Nondeterministic Finite Automaton

Nondeterministic Finite Automaton adalah sebuah *state machine* yang memiliki sekumpulan *state* berhingga Q, sekumpulan simbol Σ sebagai masukan transisi *state*, fungsi transisi F dari *state* $q \in Q$ dengan masukan simbol $\alpha \in \Sigma$ menuju *state* $Q' \subseteq Q$, sekumpulan state $A \subseteq Q$ sebagai *state* NFA selesai, dan sebuah *state* q_0 sebagai *state* NFA dimulai [1].

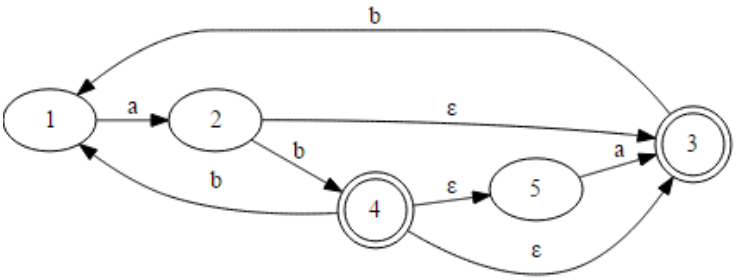
NFA dapat direpresentasikan dengan sebuah tabel atau sebuah *graph*. **Tabel 2.3.1** merepresentasikan sebuah NFA dengan 5 *state* dengan kumpulan simbol masukan $\Sigma = \{a, b\}$. *state* awal NFA dimulai dari *state* q_0 dan memiliki dua *state* selesai yakni $A = \{3,4\}$. NFA tersebut juga dapat direpresentasikan dalam bentuk *graph* seperti pada **Gambar 2.3.1.**

NFA memiliki karakteristik khusus yakni sebuah *state* dapat berpindah ke *state* lain tanpa masukan simbol apapun yang dinotasikan dengan ε sebagai simbol masukan dalam **Tabel 2.3.1.**

Tabel 2.3.1 Representasi NFA dalam tabel

	a	b	ε
q_0	2	-	-
2	-	4	3
{3}	-	1	-
{4}	-	1	3,5
5	3	-	-

Perpindahan dengan tanpa masukan simbol disebut dengan perpindahan/transisi epsilon. Dengan adanya transisi epsilon dalam NFA, satu masukan simbol dapat mengaktifkan lebih dari satu *state*. Sebagai contoh, apabila *state* aktif saat ini adalah state 2, dengan masukan simbol "b" maka beberapa *state* yang aktif selanjutnya adalah $Q' = \{4,1\}$.



Gambar 2.3.1 Representasi NFA dalam *directed graph*

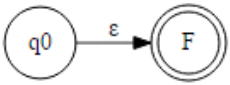
Hal ini membuat NFA menjadi sangat lambat ketika digunakan untuk melakukan pencarian *string*, terlebih jika NFA memiliki banyak *state*. Lambatnya NFA dalam melakukan pencarian *string* ini dikarenakan diperlukan penyimpanan setiap jalur *state* yang mungkin dikunjungi untuk bisa menyatakan suatu rangkaian simbol masukan dapat mencapai *state* selesai atau tidak.

2.4 Metode Thompson

Metode Thompson adalah sebuah metode yang digunakan untuk melakukan konversi dari *regular expression* ke dalam bentuk *Nondeterministic Finite Automaton* (NFA). Metode ini merupakan metode yang sederhana dan akan menghasilkan maksimal $2m$ *states* dan maksimal $4m$ transisi dimana m adalah banyak karakter alfabet dalam *regular expression*.

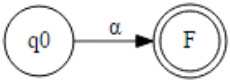
Metode ini menggunakan *automaton* yang spesifik untuk setiap operator yang digunakan dalam *regular expression*. Beberapa *automaton* tersebut adalah [1]

- *Automaton* untuk transisi epsilon ditunjukkan pada **Gambar 2.4.1**. *Automaton* ini berarti perpindahan state dari $q0$ ke F bisa terjadi tanpa masukan simbol apapun.



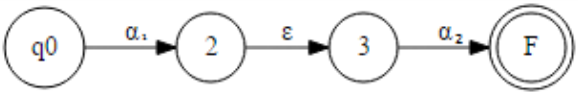
Gambar 2.4.1 *Automaton* transisi epsilon

- *Automaton* untuk sebuah masukan karakter ditunjukkan pada **Gambar 2.4.2**. *Automaton* ini berarti perpindahan state dari $q0$ ke F terjadi apabila karakter α dimasukkan sebagai simbol masukan ke NFA saat $q0$ sebagai *state* aktif.



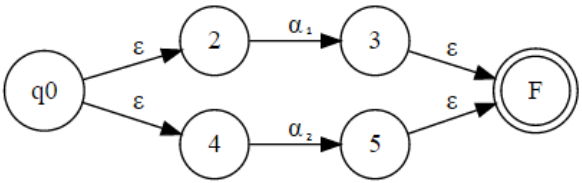
Gambar 2.4.2 *Automaton* dengan sebuah masukan karakter

- Automaton untuk operator "·" (*Concatenate*) ditunjukkan pada **Gambar 2.4.3**. *Automaton* ini berarti perpindahan *state* dari $q0$ ke F terjadi apabila NFA diberikan dua kali masukan simbol secara berurutan. untuk berpindah dari *state* q0 ke *state* 2 dengan masukan α_1, dan dari *state* 3 ke *state* F dengan masukan α_2



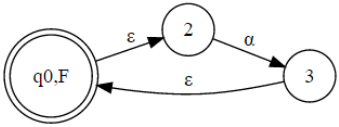
Gambar 2.4.3 *Automaton* operator *concatenate*

- *Automaton* untuk operator "|" (*Union*) ditunjukkan pada **Gambar 2.4.4**. *Automaton* ini berarti perpindahan *state* dari $q0$ ke F dapat terjadi apabila masukan yang diberikan merupakan salah satu dari simbol α_1 atau α_2.



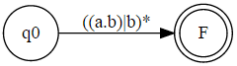
Gambar 2.4.4 *Automaton* operator Union

- *Automaton* untuk operator "*" (*Klenee star*) ditunjukkan pada **Gambar 2.4.5**. *Automaton* berarti perpindahan *state* dari $q0$ ke F dapat terjadi dengan tanpa masukan karena $q0$ adalah *state* yang sama dengan F atau dengan masukan simbol α.

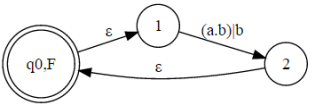


Gambar 2.4.5 *Automaton* operator Klenee star

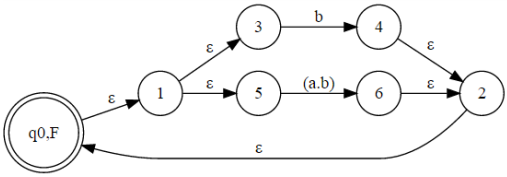
Dengan metode di atas contoh *regular expression* $\left(((a.b)|b)*\right)$ akan terbentuk NFA dengan melakukan operasi yang sudah didefinisikan oleh metode tersebut dengan proses yang ditunjukan secara berurutan pada **Gambar 2.4.6, Gambar 2.4.7, Gambar 2.4.8,** dan **Gambar 2.4.9**.



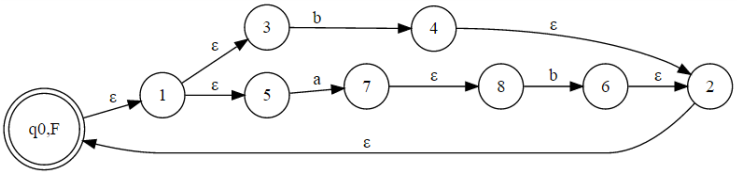
Gambar 2.4.6 *Automaton* awal yang terbentuk



Gambar 2.4.7 *Automaton* setelah operator *Klenee star*



Gambar 2.4.8 *Automaton* setelah operator *Union*



Gambar 2.4.9 NFA representasi dari *regular expression* $\left(((a.b)|b)*\right)$

2.5 Deterministic Finite Automaton

Deterministic Finite Automaton (DFA) adalah sebuah *state machine* spesialisasi dari NFA di mana DFA memiliki properti khusus yang harus dipenuhi, yaitu,

- Tidak ada transisi epsilon pada *state machine.*
- Setiap masukan simbol pada suatu *state* akan menghasilkan perpindahan ke *state* unik yang lain, dengan arti untuk masukan simbol yang berbeda maka *state* selanjutnya akan berbeda pula.

2.6 Metode Subset Construction

Karena setiap DFA merupakan NFA, maka setiap NFA yang terbentuk dari hasil konversi *regular expression* akan bisa

dikonversi ke DFA yang sepadan. Konversi NFA ke DFA dilakukan dengan menggabungkan beberapa *states* pada NFA menjadi sebuah *states* pada DFA. Penggabungan ini dapat dilakukan karena dua alasan yaitu,

- Pada NFA apabila ada dua atau lebih *state* yang terhubung dengan transisi epsilon, maka beberapa *state* tersebut dapat dianggap satu *state* pada DFA karena untuk berpindah dari satu *state* ke *state* lain dengan transisi epsilon tidak diperlukan adanya masukan simbol.
- Pada NFA apabila terdapat sebuah *state* yang apabila diberi sebuah masukan dapat berpindah ke beberapa *state* yang lain maka kumpulan *state* tersebut dapat dianggap sebuah state pada DFA.

Dari dua alasan tersebut maka didefinisikanlah dua fungsi yang akan digunakan sebagai operasi penggabungan beberapa *state* NFA menjadi *sebuah state* DFA. Dua fungsi tersebut adalah,

- Move Closure
  Move closure adalah fungsi yang digunakan untuk menemukan sekumpulan *state* pada NFA yang bisa dikunjungi dari sebuah state NFA dengan suatu masukan simbol tertentu.
- Epsilon Closure
  Epsilon Closure adalah fungsi yang digunakan untuk menemukan sekumpulan *state* pada NFA yang bisa dikunjungi dari sebuah state NFA tanpa diperlukan adanya masukan simbol tertentu.

Untuk melakukan konversi DFA dari NFA ada beberapa langkah yang perlu dilakukan dengan menggunakan fungsi *move closure* dan *epsilon closure* yang telah didefinisikan. Langkah-langkah tersebut adalah,

1. Membuat *state* awal DFA dengan melakukan *epsilon closure* pada *state* awal NFA. *state* awal DFA merupakan kumpulan *state* NFA yang dapat dikunjungi dengan transisi epsilon dari *state* awal NFA.

2. Untuk setiap *state* DFA yang baru terbentuk dilakukan langkah-langkah berikut untuk semua $\alpha \in \Sigma$.
    a. Melakukan *move closure* dengan parameter *state* DFA yang berupa kumpulan satu atau lebih *state* NFA dan masukan simbol tertentu untuk mendapatkan kumpulan *state* NFA yang dapat dikunjungi dari *state* DFA dengan masukan simbol tertentu.
    b. Melakukan *epsilon closure* pada kumpulan *state* NFA yang didapatkan pada langkah 2a untuk mendapatkan *state* baru DFA (*state* baru ini merupakan satu atau lebih *state* NFA yang dikombinasikan menjadi satu *state* DFA). Apabila pada langkah 2a tidak ada *state* NFA yang dihasilkan tidak ada *state* DFA baru yang dihasilkan.
3. Apabila masih ada state DFA baru yang terbentuk pada langkah 2, maka langkah 2 dilakukan hingga tidak ada *state* baru yang terbentuk. Apabila telah tidak ada *state* baru yang terbentuk maka dilanjutkan ke langkah 4.
4. DFA selesai dibentuk, dan *state* selesai DFA merupakan *state* DFA yang terbentuk dari *state* selesai NFA [2].

2.7 Perpangkatan Matriks

Perpangkatan matriks atau *Matrix Exponentiation* digunakan untuk mencari jumlah kemungkinan *string* yang dapat diterima oleh *regular expression* dengan memanfaatkan properti adjacency matriks yang memberikan jumlah *path* dari i ke j dengan tepat k langkah tertentu, dimana k adalah pangkat dari perpangkatan matriks. Adjacency matriks dibangun dari DFA yang dihasilkan pada proses *subset construction*.

Perpangkatan matriks dengan dimensi D dengan pangkat P dapat dilakukan dengan melakukan perkalian matriks sebanyak N kali. Pada proses perpangkatan tersebut akan terjadi

operasi perkalian sebanyak $D^2 x\, N$. Kompleksitas big O dari proses tersebut akan memiliki orde 3.

Teknik mempercepat perpangkatan matriks diantaranya adalah melakukan *Exponentiation by Squaring* atau melakukan perpangkatan dengan mengkuadratkan. Teknik ini mengurangi proses perkalian menjadi $D^2 x \log_2 N$. Teknik ini secara rekursif didefinisikan pada persamaan 1 [3].

$$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}}, apabila\ n\ ganjil \\ (x^2)^{\frac{n}{2}}, apabila\ n\ genap \end{cases} \quad (1)$$

2.8 Permasalahan CTSTRING di SPOJ

A regular expression is used to describe a set of strings. For this problem the alphabet is limited to 'a' and 'b'. R is a regular expression if:

1) R is "a" or "b"

2) R is of the form "(R1R2)" where R1 and R2 are regular expressions

3) R is of the form "(R1|R2)" where R1 and R2 are regular expressions

4) R is of the form "(R1*)" where R1 is a regular expression.

The set of strings recognised by R are as follows:

1) If R is "a", then the set of strings recognised = {a}

2) If R is "b", then the set of strings recognised = {b}

3) if R is of the form "(R1R2)" then the set of strings recognised = all strings which can be obtained by a concatenation of strings s1 and s2 where s1 is recognised by R1 and s2 by R2.

4) if R is of the form "(R1|R2)" then the set of strings recognised = union of the set of strings recognised by R1 and R2.

5) If R is of the form "(R1*)" then the the strings recognised are the empty string and the concatenation of an arbitrary number of copies of any string recognised by R1.

Given a regular expression and an integer L, you need to count how many strings of length L are recognized by it.

Gambar 2.8.1 Deskripsi permasalahan CTSTRING

Permasalahan representasi *regular expression* menjadi ke NFA atau DFA juga terdapat pada situs *Online Judge* SPOJ yang mempunyai kode permasalahan CTSTRING. Deskripsi dari permasalahan ditunjukkan pada **Gambar 2.8.1** [4]. Pada **Gambar 2.8.1** dijelaskan bahwa diberikan sebuah *regular expression* RE dan sebuah bilangan bulat L, dengan batasan *regular expression* memiliki tiga operator yakni *concatenate*, *union*, dan *klenee star* dan simbol $\Sigma = \{a, b\}$. Dari masukan

yang diberikan, hitung banyaknya *string* dengan panjang L yang dapat di terima oleh *regular expression* RE.

Berikut merupakan Format masukan pada permasalahan tersebut,

1. Baris pertama dari masukan adalah sebuah bilangan bulat N yang menyatakan berapa banyak kasus uji yang diberikan.
2. N Baris berikutnya merupakan masukan kasus uji dengan format *string* RE dan bilangan bulat L.

Format keluaran pada permasalahan tersebut adalah N buah bilangan T di mana T adalah banyaknya *string* dengan panjang L yang dapat diterima oleh *regular expression* RE. Karena bilangan T bisa sangat besar, maka T ditampilkan setelah dilakukan operasi modulus terhadap T yakni $T = T \ mod \ (10^9 + 7)$. Contoh masukan dan keluaran juga dapat dilihat pada **Gambar 2.8.2.**

Sample Input:

3

((ab)|(ba)) 2

((a|b)*) 5

((a*)(b(a*))) 100

Sample Output:

2

32

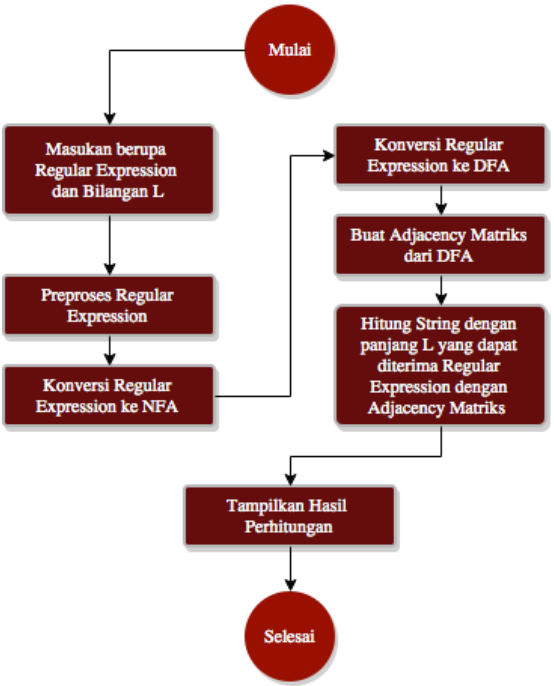
100

Gambar 2.8.2 Contoh masukan dan keluaran permasalahan

Adapun batasan dari permasalahan tersebut adalah,

1. $1 \leq N \leq 50$.
2. $1 \leq panjang(RE) \leq 100$.
3. $1 \leq L \leq 10^9$.
4. Dinilai dalam lingkungan penilaian Intel G860 3Ghz
5. Batasan waktu 0.1 detik – 0.150 detik.
6. Batasan memori 1536 MB.

2.9 Desain Umum Penyelesaian Permasalahan CTSTRING



Gambar 2.9.1 Diagram Alur penyelesaian permasalahan

Untuk menyelesaikan permasalahan studi kasus, tahap-tahap yang ditempuh dapat dilihat pada **Gambar 2.9.1**. Pada tahap awal setelah masukan diperoleh diperlukan preproses untuk memberikan operator concatenate yang tidak diberikan pada masukan studi kasus. Kemudian hasil preproses tersebut diproses untuk dikonversi menjadi NFA dengan model NFA Thompson. Usai NFA terbentuk, dilakukan konversi NFA menjadi DFA dengan metode *Subset Construction*, dan pada tahap akhir dibentuk sebuah *Adjacency Matrix* dari DFA untuk kemudian digunakan sebagai sarana perhitungan jumlah *string* yang dapat diterima oleh *regular expression* yang diberikan.

2.10 Ilustrasi Penyelesaian permasalahan CTSTRING

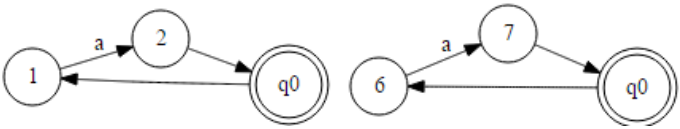
Pada permasalahan CTSTRING diberikan dua masukan yaitu *regular expression RE* dan bilangan *L*. Contoh yang akan dibahas diasumsikan dengan masukan sebagai berikut, $RE = \Big((a\,*)\big(b(a\,*)\big)\Big)$ dan $L = 100$.

Dari kedua masukan tersebut, hal yang pertama dilakukan adalah melakukan preproses terhadap *RE* dikarenakan tidak terdapat operator *concatenate* yang diberikan secara eksplisit dalam format masukan dari Sphere Online Judge. Usai preproses dilakukan makan *regular expression* yang terbentuk menjadi $RE = \Big((a\,*) + \big(b + (a\,*)\big)\Big)$.

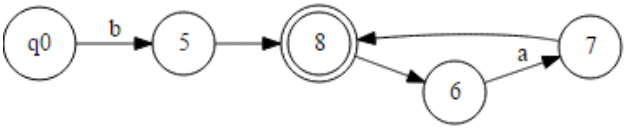
Proses konversi *regular expression* ke NFA dilakukan setelah preproses. Proses ini dilakukan dengan melakukan iterasi setiap karakter *RE* hasil preproses. Pada proses iterasi akan dibentuk kombinasi *automaton* untuk setiap evaluasi operator. Pada contoh di atas akan ada 4 evaluasi operator yang akan dilakukan, yakni dua kali "*", dan dua kali "+".

Evaluasi pertama pertama dan kedua adalah $(a\,*)$, ketiga adalah $\big((b + (a\,*)\big)$, dan terakhir $\Big((a\,*) + \big(b + (a\,*)\big)\Big)$. Ilustrasi **Gambar 2.10.1**, **Gambar 2.10.2**, dan **Gambar 2.10.3** adalah tahap-tahap pembentukan NFA untuk setiap operasi yang dilakukan. Operasi tersebut dilakukan dengan mengikuti metode Thompson seperti dijelaskan pada subbab **2.4**.

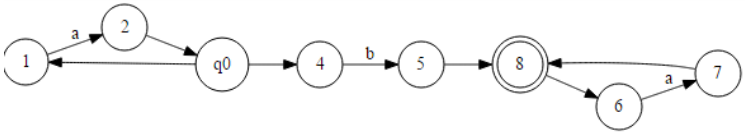
Hasil akhir dari tahap tersebut (**Gambar 2.10.3**) merupakan NFA yang kemudian digunakan untuk konversi menjadi DFA. Proses konversi DFA dilakukan dengan proses yang telah dijelaskan pada subbab **2.6**. Inisialisasi awal DFA dengan melakukan *EpsilonClosure* dari awal DFA yakni *state* 9.



Gambar 2.10.1 Evaluasi operasi (a*) pertama dan kedua



Gambar 2.10.2 Evaluasi operasi (b+(a*))



Gambar 2.10.3 Evaluasi operasi ((a*)+(b+(a*)))

State awal DFA merupakan kombinasi dari beberapa *state* NFA yang dapat dicapai dengan transisi epsilon, yakni *state* 1,3, dan 4. Dari *state* awal yang didapatkan, kemudian dilakukan iterasi untuk memperoleh DFA seutuhnya, iterasi yang dilakukan dapat dilihat pada **Tabel 2.10.1**.

Tabel 2.10.1 Iterasi dalam konversi NFA menjadi DFA

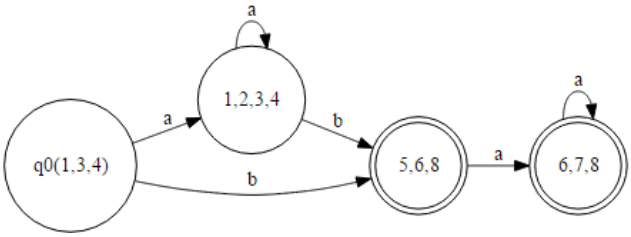
Iterasi	DFA *state*	Masukan Alfabet	
		aε	**bε**
1	1,3,4	1,2,3,4	5,6,8
2	1,2,3,4	1,2,3,4	5,6,8
3	5,6,8	6,7,8	-
4	6,7,8	6,7,8	-

Selesai iterasi untuk membangkitkan DFA dari NFA selesai maka DFA yang terbentuk dapat dilihat pada **Gambar 2.10.4**, dan **Gambar 2.10.5**.

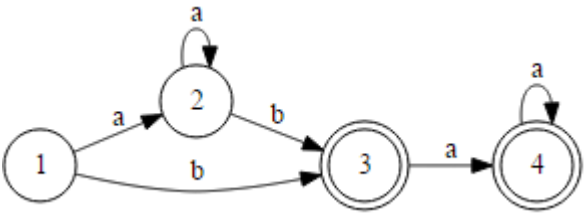
Apabila telah terbentuk DFA, maka tahap selanjutnya yang dilakukan adalah menghitung banyaknya *string* dengan

panjang *L* yang dapat diterima pada model tersebut. *String* diterima jika dari state awal DFA diberikan masukan urutan karakter alfabet pada *string* hingga seluruh karakter masukan telah berjumlah *L* dan *state* terakhir yang aktif adalah *state* akhir DFA yakni *state* 3 dan 4.

Sebagai contoh *string* s=”aaaa” akan melalui urutan *state* $1 \rightarrow 2 \rightarrow 2 \rightarrow 2 \rightarrow 2$ dan tidak berakhir pada *state* akhir DFA, sehingga *string* aaaa tidak diterima.



Gambar 2.10.4 hasil konversi NFA ke DFA



Gambar 2.10.5 DFA yang telah disederhanakan

Pada contoh sebelumnya *string* s memiliki panjang 4 karakter, tentu saja sangat mudah untuk mencoba seluruh kemungkinan, dengan alfabet={“a”,”b”} dan panjang 4 hanya diperlukan 2^4percobaan untuk menguji semua *string* dengan panjang 4. Namun pada permasalahan studi kasus yang digunakan, batas atas *L* adalah 10^9 dan tentu saja untuk mencoba seluruh kemungkinan *string* dengan panjang tersebut sangat tidak mungkin karena akan diperlukan 2^{10^9}operasi.

Untuk itu perlu dibuat *adjacency matrix* untuk DFA yang notabene juga merupakan *directed graph*. *Adjacency matrix* untuk sebuah *graph* merepresentasikan relasi tiap *vertex/states*. Dari DFA pada **Gambar 2.10.5** *adjacency matrix* yang dihasilkan dapat dilihat pada **Gambar 2.10.6.**

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Gambar 2.10.6 *Adjacency matrix* dari DFA yang dibuat

Dari matriks tersebut dapat dilihat relasi antar *state* sebagai contoh pada baris pertama {0,1,1,0} yang memiliki arti *state* 1 memiliki hubungan ke *state* 2, dan 3 dengan satu langkah perpindahan, untuk dua langkah perpindahan maka *adjacency matrix* yang dihasilkan dapat dilihat pada **Gambar 2.10.7.**

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Gambar 2.10.7 *Adjacency matrix* untuk perpindahan sebanyak 2 langkah

Matriks pada gambar **Gambar 2.10.7** diperoleh dengan memangkatkan matriks pada **Gambar 2.10.6** dengan pangkat 2. Untuk memperoleh *string* yang dapat diterima oleh *regular expression* dengan panjang $L = 100$ hal yang perlu dilakukan adalah memangkatkan matriks pada **Gambar 2.10.6** dengan 100. Hasil yang diperoleh dapat dilihat pada **Gambar 2.10.8**.

Dari **Gambar 2.10.8** untuk memperoleh berapa banyak *string* dengan panjang 100 yang diterima, maka dilakukan penjumlahan untuk setiap *cell* yang merepresentasikan relasi *state* awal DFA dan *state* akhir DFA, yakni baris pertama kolom ketiga, dan baris pertama kolom keempat. Hasil yang didapatkan adalah 100.

$$\begin{bmatrix} 0 & 1 & 1 & 99 \\ 0 & 1 & 1 & 99 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Gambar 2.10.8 *Adjacency matrix* setelah dipangkatkan 100

Jadi dapat disimpulkan, dengan *regular expression* $RE = \Big((a*) + \big(b + (a*)\big)\Big)$ dan $L = 100$ ada 100 *string* dengan panjang 100 yang diterima oleh *regular expression RE*.

[Halaman ini sengaja dikosongkan]

BAB III
DESAIN PERANGKAT LUNAK

Pada bab ini penulis menjelaskan tentang Desain metode yang digunakan untuk menyelesaikan permasalahan pada Tugas Akhir.

3.1 Desain Sistem Secara Umum

Sistem yang dibuat akan menerima masukan berupa sebuah bilangan N pada awal sistem bekerja kemudian N masukan berikutnya berisikan *regular expression* RE dan bilangan L. Keluaran yang diharapkan dari sistem adalah bilangan T yang merupakan banyaknya *string* dengan panjang L yang dapat diterima oleh RE.

Pada setiap kasus uji, sistem akan melakukan *preprocess* pada RE untuk memberikan penanda operator *concatenate* dan kemudian sistem akan memulai untuk membaca RE. Saat pembacaan RE sistem akan membentuk NFA berdasar alfabet dan operator yang dibaca.

```
main()
1.  input N
2.  for i=1 to N
3.   initializeVariables()
4.   input RE, L
5.   RE := Preprocess(RE)
6.   NFA := ConvertREtoNFA(RE)
7.   DFA := ConvertNFAtoDFA(NFA)
8.   adjM := Create adjacency matrix from DFA
9.   MatrixPower(adjM, L)
10.  sum cell of the matrix which have finish
     state of NFA
11.  Output the sum
```

Gambar 3.1.1 *Pseudocode* fungsi main

Usai NFA terbentuk akan dilakukan konversi dari NFA ke DFA. DFA merupakan sebuah *graph* berarah yang dapat

dimodelkan menjadi *adjcacency matrix* untuk melakukan menghitung T, akan dilakukan perpangkatan *matrix* sehingga hasil T merupakan jumlah bilangan pada *matrix* yang ada pada kolom *state* selesai pada sebuah NFA. *Pseudocode* fungsi utama program dapat dilihat pada **Gambar 3.1.1**.

3.2 Desain Algoritma

Sistem terdiri dari beberapa fungsi utama yaitu Preprocess, ConvertREtoNFA, ConvertNFAtoDFA, dan MatrixPower. Pada subbab ini akan dijelaskan tentang masing-masing fungsi utama dan *pseudocode* dari fungsi-fungsi tersebut.

3.2.1 Desain fungsi Preprocess

Fungsi Preprocess digunakan untuk memberikan operator *concatenate* pada *string regular expression*. Fungsi Preprocess dibutuhkan karena pada format masukan, operator *concatenate* tidak memiliki simbol sehingga dapat menimbulkan ambiguitas. *Pseudocode* fungsi Preprocess dapat dilihat pada **Gambar 3.2.1**.

```
/*
* RE   unpreprocessed regular expression
*
* return value
* retRE      preprocessed regular expression
*/
retRE := Preprocess(RE)
```

```
1.  for i=0 to length(RE)
2.   if (RE[i] is alphabet or '(') and (RE[i-1]
    is alphabet or ')')
3.     insert operator "." to RE[i+1]
4,  return RE
```

Gambar 3.2.1 *Pseudocode* fungsi preprocess

Fungsi Preprocess dilakukan dengan menambah operator "." pada indeks di mana terdapat kemungkinan kasus seperti berikut

- "aa", "ab", "ba", "bb".
- ")a", ")b".
- "a(", "b(".
- dan ")(".

Kasus-kasus tersebut diimplementasikan dalam fungsi preprocess pada baris ke 2.

3.2.2 Desain fungsi ConvertREtoNFA

Fungsi ConvertREtoNFA digunakan untuk melakukan konversi *regular expression* ke NFA. Fungsi ini mengimplementasikan model *Automaton* pada metode Thompson. *Pseudocode* fungsi ini dapat dilihat pada **Gambar 3.2.2**. Fungsi ini berjalan dengan membaca *regular expression* per karakter (baris 1) dan akan mengevaluasi sebuah *regular expression* untuk setiap pasang karakter "(" dan ")" (baris 6). Untuk melakukan evaluasi tersebut dibutuhkan dua buah *stack*, yakni *stack* yang menyimpan *automaton* NFA sementara, dan *stack* operator yang menyimpan operator yang harus dievaluasi saat karakter ")" ditemui. Untuk setiap alfabet yang ditemui dalam pembacaan *regular expression* akan dibuat *automaton* seperti pada **Gambar 2.4.2** (baris 4). Saat karakter ")" ditemui dalam pembacaan *regular expression*, akan dilakukan evaluasi terhadap operator yang digunakan dalam operasi selanjutnya dengan mengambil operator teratas pada *stack* operator (baris 7 dan 8).

Automaton baru akan dibuat setiap operator dievaluasi, apabila operator *concatenate* ditemui, maka akan diambil dua *automaton* sementara dari *stack states* yang kemudian digunakan untuk melakukan operasi penggabungan sehingga dihasilkan *automaton* baru untuk disimpan ke *stack states* (baris 9 s.d. 11). Proses yang sama juga berlaku untuk operator *union* dengan melakukan operasi *union* untuk menghasilkan

automaton baru (baris 12 s.d. 14). Pada saat melakukan operasi *klenee star* hanya ada satu *automaton* yang diambil dari *stack states* yang kemudian dimodifikasi sehingga membentuk *automaton klenee star* (baris 15 s.d. 17).

```
* RE   preprocessed regular expression
* NFA  finite state machine model that has
*      epsilon transition
NFA := ConvertREtoNFA(RE)
```

```
1.  for i=1 to length(RE)
2.    if RE[i] is '(' continue loop
3.    else if RE[i] is alphabet
4.      create new NFA automaton with alphabet
    as transition symbol
5.      push newly created NFA automaton to
    states
6.    else if RE[i] is ')'
7.     if stack operator is not empty
8.      pop 1 from stack operator to op
9.     if op is concatenate operator
10.     pop 2 automaton from stack states to st1
    and st2 then create new automaton by st1
    concat st2
11.     push newly created automaton to states
12.    else if op is union operator
13.     pop 2 automaton from stack states to st1
    and st2 then create new automaton by st1
    union st2
14.     push newly created automaton to states
15.    else if op is klenee star operator
16.     pop 1 automaton from stack to st1 and
    create new automaton by modifying st1 to
    macth klenee star Thompson model
17.     push newly created automaton to states
18.   else if RE[i] is operator
19.    push operator to operator stack
20  return NFA = pop last automaton from states
    stack after for loop finished
```

Gambar 3.2.2 *Pseudocode* fungsi ConvertREtoNFA

Apabila karakter yang ditemui dalam *regular expression* adalah operator yakni karakter "|", ".", dan "*",

maka karakter tersebut akan disimpan ke *stack* operator (baris 18 dan 19). Setelah seluruh karakter *regular expression* selesai dibaca dan dievaluasi akan tersisa sebuah *automaton* yang ada pada *stack states*. *Automaton* ini adalah *automaton* NFA hasil konversi dari RE (baris 20).

3.2.3 Desain fungsi ConvertNFAtoDFA

```
* NFA  finite state machine model that has
*      epsilon transition
* return value
* DFA  finite state machine model that has no
*      epsilon transition
DFA = ConvertNFAtoDFA(NFA)
```

```
1.  DFA := create empty state machine
2.  NFAinitialState = NFA.getInitialstate()
3.  St := EpsilonClosure(NFAinitialState)
4.  DFASt := CreateDFAState(St, &DFA)
5.  push DFASt to stack tmp
6.  while tmp is not empty
7.   pop tmp and assign to dv
8.   for i=0 to sizeof(alphabet)
9.    St := MoveClosure(dv,alphabet[i])
10.   St := EpsilonClosure(St)
11.   if St is empty set
12.    continue loop
13.   else
14.    DFASt := CreateDFAState(St, &DFA)
15.    Assign alphabet[i] as transition symbol
    from dv to DFASt
16.    If DFAst not already created before
17.     push DFAst to stack tmp
18. return DFA
```

Gambar 3.2.3 *Pseudocode* fungsi ConvertNFAtoDFA

Fungsi ConvertNFAtoDFA merupakan fungsi yang digunakan untuk melakukan konversi NFA menjadi DFA. Fungsi ini memiliki beberapa fungsi pendukung diantaranya adalah fungsi EpsilonClosure, MoveClosure, dan CreateDFAState. *Pseudocode* fungsi ConvertNFAtoDFA dapat dilihat pada **Gambar 3.2.3.** Fungsi ini dimulai dengan membuat

state awal DFA di mana *state* tersebut merupakan *state* awal NFA dan kumpulan *state* NFA yang dapat dicapai dari *state* awal NFA melalui transisi epsilon (baris 3 dan 4). Setelah *state* awal DFA terbentuk tahapan yang dilakukan selanjutnya adalah melakukan metode *Subset construction* yang telah dijelaskan pada subbab **2.6** halaman 11, pada *pseudocode* **Gambar 3.2.3** hal tersebut dilakukan pada baris 6 s.d 17.

Fungsi EpsilonClosure merupakan salah satu fungsi pendukung dalam proses pembuatan DFA. *Pseudocode* fungsi ini dapat dilihat pada **Gambar 3.2.4**. Fungsi ini memiliki sebuah parameter, yakni kumpulan *state* NFA yang saat ini sedang aktif. Fungsi tersebut diawali dengan membuat daftar pengecekan *state* NFA yang saat ini sedang aktif (baris 1 s.d. 3). Kemudian untuk setiap *state* NFA yang sedang aktif tersebut akan dilakukan pengecekan terhadap *state* aktif selanjutnya yang dapat dikunjungi dengan transisi epsilon (baris 4 s.d 6). Apabila dari *state* yang baru dikunjungi setelah dilakukan transisi epsilon masih memiliki *state* lain yang dapat dikunjungi dengan transisi epsilon maka *state* baru tersebut akan dimasukkan ke daftar pengecekan dan daftar *state* yang dapat dikunjungi dengan transisi epsilon untuk kemudian dilakukan pengecekan (baris 8 s.d 10). Pengecekan tersebut dilakukan selama daftar pengecekan masih belum seluruhnya dilakukan pengecekan (baris 4). Apabila seluruh daftar telah selesai dilakukan pengecekan maka daftar *state* yang dapat dikunjungi dengan transisi epsilon tersebut dikembalikan ke pemanggil fungsi EpsilonClosure.

Fungsi pendukung lain dalam konversi NFA ke DFA adalah fungsi MoveClosure. *Pseudocode* fungsi ini dapat dilihat pada **Gambar 3.2.5**. Fungsi ini menerima kumpulan *state* aktif sebagai parameter dan sebuah alfabet. Untuk setiap *state* aktif akan dilakukan pengecekan apakah ada *state* lain yang dapat dikunjungi dari state yang sedang aktif dengan masukan alfabet dari parameter fungsi (baris 1 dan 2).

```
* St          set of states to transite from
* retSt       set of states that can be reached
*             from St with epsilon transition
retSt = EpsilonClosure(St)
1.  for i=0 to sizeof(St)
2.   add st to retSt
3.   push st to stack tmp
4.  while stack tmp is not empty
5.   pop tmp and assign to t
6.   r:=next state that can be reached from t
     with epsilon transition
7.   for i=0 to r
8.    if t.next[i] not exist in retSt
9.     add t.next[i] to retSt
10.    push t.next[i] to stack tmp
11. return retSt
```

Gambar 3.2.4 *Pseudocode* fungsi EpsilonClosure

```
* St          set of states to transite from
* alphabet    symbol input to state so the
*             trasition can happen from a state
*             to another set of states
* retSt       set of states that can be reached
*             from St with input alphabet
retSt = MoveClosure(St, alphabet)
1.  for i=0 to sizeof(St)
2.   if St.next(alphabet) is not empty
3.    add St.next(alphabet) to retSt
4.  return retSt
```

Gambar 3.2.5 *Pseudocode* fungsi MoveClosure

Apabila ada *state* yang dapat dikunjungi dengan masukan alphabet tersebut, maka *state* tersebut dimasukan ke daftar state yang dapat dikunjungi (baris 3). Setelah seluruh *state* aktif selesai dilakukan pengecekan maka daftar *state* yang dapat dikunjungi dikembalikan ke pemanggil fungsi MoveClosure (baris 4).

Fungsi pendukung terakhir dalam konversi NFA ke DFA adalah CreateDFAState. *Pseudocode* fungsi ini dapat dilihat pada **Gambar 3.2.6.** Fungsi ini digunakan untuk membuat dan menambahkan *state* yang baru terbentuk dalam proses pembuatan DFA ke dalam *state machine* DFA. Dalam fungsi

ini juga dilakukan pengecekan apakah state DFA yang akan ditambahkan sudah pernah dibuat sebelumnya atau belum (baris 1). Apabila sudah pernah dibuat, maka fungsi ini hanya akan mengembalikan *state* yang sudah ada (baris 2). Apabila belum, maka fungsi ini akan membuat state DFA kemudian menambahkannya sebelum dikembalikan ke pemanggil fungsi CreateDFAState (baris 4).

```
/*
* St          set of states to join as one DFA
*             state
* DFA         DFA state machine
*
*return value
* DFASt       DFA state
*/
DFASt = CreateDFAState(St, &DFA)
```

```
1.  if St already created and exists in DFA
2.   return St
3.  else add St to DFA
4.  return St
```

Gambar 3.2.6 *Pseudocode* fungsi CreateDFAState

3.2.4 Desain fungsi MatrixPower

```
* AdjM        adjacency matrix
* L
power
* retAdjM     Lth power Matrix
retAdjM = MatrixPOwer(AdjM, L)
```

```
1.  Create identity matrix I that has same
    dimension as adjM
3.  while L>0
4.   if L is odd
5.    I := AdjM x I
6.   AdjM := AdjM x AdjM
7.   L := L/2
8.  return I
```

Gambar 3.2.7 *Pseudocode* fungsi MatrixPower

Fungsi MatrixPower memiliki dua parameter yakni matriks AdjM dan bilangan L. Fungsi ini digunakan untuk memangkatkan matriks AdjM ke pangkat L. *pseudocode* fungsi ini dapat dilihat pada **Gambar 3.2.7**. Fungsi ini menggunakan teknik *Exponentiation by Squaring* seperti yang dijelaskan pada subbab **2.7** halaman 13.

3.3 Desain Struktur Data

Sistem yang dibuat membutuhkan beberapa struktur data untuk menyimpan beberapa data penting. Struktur data pertama yang dibutuhkan adalah struktur data yang dapat menyimpan sekumpulan data tanpa duplikasi dan dapat dilakukan pengecekan secara cepat untuk kesamaan isi himpunan. Struktur data ini dibutuhkan untuk memberikan label yang berbeda untuk tiap *state* DFA. Di mana *state* DFA merupakan himpunan satu atau lebih *state* NFA. Struktur data yang tepat digunakan untuk kasus tersebut adalah struktur data *Set*. Hal ini dikarenakan struktur data *Set* dapat melakukan penyimpanan data dan dapat dipastikan tidak ada duplikasi pada data yang disimpan.

Struktur data kedua yang dibutuhkan adalah struktur data yang dapat digunakan untuk mengakses data dengan parameter tertentu yang diberikan. Struktur data ini digunakan dalam penyimpanan *automaton* DFA untuk memperoleh *state* tertentu dengan parameter himpunan satu atau lebih *state* NFA yang kemudian disimpan dalam struktur data *Set*. Struktur data yang tepat untuk permasalahan seperti ini adalah *Dictionary/Map* di mana suatu data dapat diakses berdasar *key* yang diberikan. Untuk keperluan di atas maka *Dictionary* akan menggunakan *Set state* NFA sebagai *key* dan *state* DFA sebagai data yang disimpan.

Struktur data ketiga yang dibutuhkan adalah struktur data yang dapat digunakan untuk mensimulasikan proses *Last In First Out*(LIFO). Struktur data ini digunakan untuk memproses *regular expression* yang berbentuk operasi infix.

Selain itu struktur data ini diperlukan untuk melakukan proses *traversal* yakni mengunjungi *state* dalam konversi NFA menjadi DFA. Struktur data yang tepat untuk permasalahan tersebut adalah *stack* di mana data juga dilakukan penyimpanan dengan kaidah LIFO.

Struktur data terakhir yang dibutuhkan adalah struktur data yang dapat menyimpan sepasang data sekaligus. Struktur data ini dibutuhkan untuk menyimpan dan memonitor pasangan *state* awal NFA dan *state* akhir NFA saat proses konversi *regular expression* ke NFA. Struktur data ini dapat dipenuhi oleh struktur data *Pair*.

3.4 Desain Pembangkit Kasus untuk Uji Kinerja.

Untuk melakukan uji kinerja diperlukan sebuah program pembantu yang dapat melakukan pembangkitan kasus uji sebagai masukan program Tugas Akhir. *Pseudocode* program pembangkit kasus uji dapat dilihat pada **Gambar 3.4.2**, dan **Gambar 3.4.1**.

Fungsi expand pada **Gambar 3.4.2** digunakan untuk menambah sebuah operasi regular expression baru dengan menyisipkan sebuah operasi ke *index* yang acak. Fungsi akan mengembalikan *string* hasil penambahan operasi ke fungsi utama pada **Gambar 3.4.1**. pada fungsi utama, akan dipanggil fungsi Expand hingga *string* mencapai batas karakter tertentu.

```
main()
1.  Initialize random seed()
2.  print(number of test case)
3.  for i=0 to number of test case
4.   initialize str = "a";
5.   while length of str < 100
6.    index = rand_int() between [1…length(str)]
7.    str = Expand(str)
8.   print(str+" "+rand_int() between 1…10^9)
```

Gambar 3.4.1 *Pseudocode* fungsi utama pembangkit kasus uji

```
* index       integer that represent position in
*             strIN
* strIN       string before expansion.
* str         Regex string after expansion
str = Expand(index, str)
```

```
1.  if strIN[index] is "a" or "b" change index
    to index of not "a" or "b" character
2.  str = strIN
3.  I = rand_int() with range [1…3]
4.  J = rand_int() with range [1…2]
5.  if I = 1
6.   if J = 1
7.    insert "(ab)" to str[index+1]
8.   else
9.    insert "(ba)" to str[index+1]
10. else if I = 2
11.  if J = 1
12.   insert "(a|b)" to str[index+1]
13.  else
14.   insert "(b|a)" to str[index+1]
15. else
16.  if J = 1
17.   insert (a*) to str[index+1]
18.  else
19.   insert (b*) to str[index+1]
20  return str
```

Gambar 3.4.2 *Pseudocode* fungsi Expand

Usai *string* mencapai batas karakter tertentu, kemudian akan dicetak hasil Expand terakhir yang merupakan *RE* dan sebuah bilangan *L* random pada rentang $[1…10^9]$.

[Halaman ini sengaja dikosongkan]

BAB IV
IMPLEMENTASI

Pada bab ini akan dibahas mengenai implementasi yang dilakukan berdasarkan rancangan yang telah dijabarkan pada bab sebelumnya. Implementasi kode program dilakukan dengan menggunakan bahasa C++.

4.1 Lingkungan Implementasi

Lingkungan implementasi yang akan digunakan adalah,

1. Perangkat Keras
   Processor Intel® Core™ i3-2310M CPU @ 2.10 GHz
   RAM 4 GB
   Sistem Operasi 64-bit
2. Perangkat Lunak
   Sistem Operasi Windows 8.0
   Integrated Development Environment Code::Blocks 13.12

4.2 Konstanta dan Variabel Global

Pada implementasi program, dibuat beberapa konstanta yang berlaku pada seluruh program dan beberapa variabel global. Variabel global merupakan variabel yang dapat diakses dari seluruh fungsi yang ada pada program. Hal ini dilakukan untuk memudahkan pengiriman variabel antar fungsi. Konstanta dan variabel global yang dibuat dapat dilihat pada **Kode Sumber 4.2.1**. Pada baris 1 dan 2 didefinisikan konstanta modulus dan *state* maksimal dari NFA maupun DFA. Variabel states pada baris 4 digunakan untuk menyimpan *automaton* sementara saat *state machine* NFA dibentuk dari interpretasi *regular expression*.

Variabel nfa_nodes pada baris 5 digunakan untuk menyimpan seluruh *state* NFA dan ke mana *state* tersebut akan

berubah ketika menerima masukan alfabet tertentu. Variabel dfa_graph pada baris 6 digunakan untuk menyimpan *state machine* dari DFA, variabel ini memiliki fungsi utama sebagai pencari *state* DFA tertentu apabila diketahui kumpulan *state* NFA yang menyusunnya. Variabel dfa pada baris 7 digunakan untuk menyimpan seluruh *state* DFA dan ke mana *state* tersebut akan berubah ketika menerima masukan alphabet tertentu. Variabel dfa_lbl pada baris 8 digunakan untuk menyimpan kumpulan *state* NFA yang menyusun sebuah *state* DFA. Variabel nid dan did pada baris 9 dan 10 digunakan untuk penomoran *state* yang terbentuk baik dari NFA dan DFA.

```
1.  #define MOD_CONSTANT 1000000007
2.  #define MAX_NODE 200
3.
4.  stack<pair_int> states;
5.  vector<int> nfa_nodes[MAX_NODE][3];
6.  map<dfa_label<int>, int> dfa_graph;
7.  int dfa[MAX_NODE][2];
8.  dfa_label<int> dfa_lbl[MAX_NODE];
9.  int nid = 0;
10. int did = 0;
```

Kode Sumber 4.2.1 Konstanta dan variabel global

4.3 Implementasi Fungsi Main

Fungsi main merupakan fungsi utama yang dipanggil pertama kali saat program berjalan. Fungsi ini diimplementasikan berdasarkan *pseudocode* yang dibuat pada **Gambar 3.1.1**. Implementasi fungsi main dapat dilihat pada **Kode Sumber 4.3.1**, **Kode Sumber 4.3.2**, dan **Kode Sumber 4.3.3.**

```
1.  int main()
2.  {
3.      char regexp[121], input[222];
4.      int dv, dv2, L, nfaFinish, test;
```

Kode Sumber 4.3.1 Implementasi fungsi main(1)

```
5.      pair_int state;
6.      for(scanf("%d", &test);test--;)
7.      {
8.          Initialize();
9.          scanf("%s %d", regexp, &L);
10.         Preprocess(regexp, input);
11.         ConvertREtoNFA(input);
12.         state = states.top();
13.         states.pop();
14.         nfaFinish = state.b;
15.         ConvertNFAtoDFA(state);
16.         long long** adjMatrix, **result;
17.         map<dfa_label<int>,   int>::iterator
    it_dfa;
18.         adjMatrix = (long long**)calloc(did,
    sizeof(long long*));
19.         result  =  (long  long**)calloc(did,
    sizeof(long long*));
20.         for(int i=0;i<did;i++)
21.         {
22.             adjMatrix[i]       =       (long
    long*)calloc(did, sizeof(long long));
23.             result[i]        =         (long
    long*)calloc(did, sizeof(long long));
24.         }
25.              for(it_dfa = dfa_graph.begin();
    it_dfa!=dfa_graph.end(); it_dfa++)
26.         {
27.             dv = it_dfa->second;
28.             for(int i=0;i<2;i++)
29.             {
30.                 dv2 = dfa[dv][i];
31.                 if( dv2 !=-1 )
32.                 {
33.                     adjMatrix[dv][dv2]++;
34.                 }
35.             }
36.         }
37.         MatrixPower(adjMatrix, L, result);
38.         long long res = 0;
```

Kode Sumber 4.3.2 Implementasi fungsi main(2)

```
39.     for(it_dfa=dfa_graph.begin();it_dfa    !=
40. dfa_graph.end(); it_dfa++){
41.             if(dfa_lbl[it_dfa-
    >second].count(nfaFinish))
42.             {
43.                 res    +=    result[0][it_dfa-
    >second];
44.             }
45.         }
46.         printf("%I64d\n", res%MOD_CONSTANT);
47.     }
48.     return 0;
49. }
```

Kode Sumber 4.3.3 Implementasi fungsi main(3)

Pada fungsi main terdapat pemanggilan fungsi Initialize pada baris 8. Fungsi Initialize digunakan untuk melakukan inisialisasi variabel pada setiap kasus uji yang diberikan. Implementasi fungsi Initialize dapat dilihat pada **Kode Sumber 4.3.4.**

```
1.  void Initialize()
2.  {
3.      nid = 0;
4.      did = 0;
5.      memset(dfa, -1, sizeof(dfa));
6.      memset(dfa_lbl, 0, sizeof(dfa_lbl));
7.
8.
9.      for(int i=0;i<MAX_NODE;i++)
10.     {
11.         for(int j=0;j<3;j++)
12.         {
13.             nfa_nodes[i][j].clear();
14.         }
15.     }
16.     while(!states.empty())
17.         states.pop();
18.     dfa_graph.clear();
19. }
```

Kode Sumber 4.3.4 Implementasi fungsi Initialize

4.4 Implementasi Fungsi Preprocess

Fungsi Preprocess adalah fungsi yang digunakan untuk memberikan operator *concatenate* berupa karakter "." pada *regular expression*. Fungsi Preprocess diimplementasi berdasarkan *pseudocode* pada **Gambar 3.2.1.** Implementasi fungsi tersebut dapat dilihat pada **Kode Sumber 4.4.1.**

```
1.  void Preprocess(char in[], char input[])
2.  {
3.      int len = strlen(in);
4.      int iter=0;
5.      for(int i=0;i<len; i++)
6.      {
7.          if(i!=0 && ( in[i]=='(' || in[i]=='a'
    || in[i] == 'b') && (in[i-1]==')' || in[i-
    1]=='a' || in[i-1]=='b'))
8.          {
9.              input[iter++]='.';
10.         }
11.         input[iter++]=in[i];
12.     }
13.     input[iter]=0;
14. }
```

Kode Sumber 4.4.1 Implementasi fungsi Preprocess

4.5 Implementasi Fungsi ConvertREtoNFA

Fungsi ConvertREtoNFA berfungsi untuk melakukan konversi dari *regular expression* menjadi *state machine* NFA. Fungsi ConvertREtoNFA diimplementasi berdasarkan *pseudocode* pada **Gambar 3.2.2.** Implementasi dapat dilihat pada **Kode Sumber 4.5.1, Kode Sumber 4.5.2,** dan **Kode Sumber 4.5.3.**

```
1.  void ConvertREtoNFA(char input[]){
2.      int len = strlen(input);
3.      stack<char> ops;
4.      char op;
5.      for(int i=0;i<len;i++)
```

Kode Sumber 4.5.1 Implementasi fungsi ConvertREtoNFA (1)

```
7.      {
8.          switch(input[i])
9.          {
10.         case '(':
11.                continue;
12.             break;
13.         case 'a':
14.         case 'b':
15.             {
16.                 nfa_nodes[nid][input[i]-
    'a'].push_back(nid+1);
17.                 states.push(pair_int(nid,
    nid+1));
18.                 nid+=2;
19.             }
20.             break;
21.         case ')':
22.             op = '-';
23.             if(ops.empty() == false)
24.             {
25.                 op = ops.top();
26.                 ops.pop();
27.             }
28.             switch(op)
29.             {
30.             case '|':
31.                 {
32.                     operan2 = states.top();
    states.pop();
33.                     operan1 = states.top();
    states.pop();

34. nfa_nodes[nid][2].push_back(operan1.a);

35. nfa_nodes[nid][2].push_back(operan2.a);

36. nfa_nodes[operan1.b][2].push_back(nid+1);

37. nfa_nodes[operan2.b][2].push_back(nid+1);
```

Kode Sumber 4.5.2 Implementasi fungsi ConvertREtoNFA (2)

```
38.
    states.push(pair_int(nid, nid+1));
39.                     nid+=2;
40.                 }
41.                 break;
42.             case '.':
43.                 {
44.                     operan2 = states.top();
    states.pop();
45.                     operan1 = states.top();
    states.pop();

46.  nfa_nodes[operan1.b][2].push_back(operan2.a)
    ;
47.
48.  states.push(pair_int(operan1.a, operan2.b));
49.                 }
50.                 break;
51.             case '*':
52.                 {
                        operan2 = states.top();
    states.pop();
53.
    nfa_nodes[nid][2].push_back(operan2.a);
54.
55.  nfa_nodes[operan2.b][2].push_back(nid);

56.  states.push(pair_int(nid, nid));
57.                     nid++;
58.                 }
59.                 break;
60.             }
61.             break;
62.         default:
63.             ops.push(input[i]);
64.         }
65.     }
    }
```

Kode Sumber 4.5.3 Implementasi fungsi ConvertREtoNFA (3)

4.6 Implementasi Fungsi ConvertNFAtoDFA

Fungsi ConvertNFAtoDFA berfungsi untuk melakukan konversi dari *state machine NFA* menjadi *state machine* DFA.

Fungsi tersebut diimplementasi berdasarkan *pseudocode* pada **Gambar 3.2.3.** Implementasi fungsi dapat dilihat pada **Kode Sumber 4.6.1.**

Pada fungsi ConvertNFAtoDFA terdapat tiga fungsi penunjang. Yang pertama adalah fungsi MoveClosure. Fungsi ini digunakan untuk melakukan pengecekan *state* aktif terhadap masukan alfabet tertentu. Fungsi ini berdasarkan *pseudocode* pada **Gambar 3.2.5.** Implementasi fungsi MoveClosure dapat dilihat pada **Kode Sumber 4.6.2.**

```
1.  void ConvertNFAtoDFA(pair_int state)
2.  {
3.      int dv, initDFA, NFAstart = state.a, tmp;
4.      stack<int> st;
5.      dfa_label<int> visited;
6.      dfa_label<int> retval;
7.      retval.push(NFAstart);
8.      EpsilonClosure(retval);
9.      initDFA = CreateDFAState(retval);
10.     visited.push(initDFA);
11.     st.push(initDFA);
12.     while(st.empty() == false){
13.         dv = st.top();
14.         st.pop();
15.         for(int i=0; i<2; i++)
16.         {
17.         MoveClosure((dfa_lbl[dv]),i,retval);
18.             EpsilonClosure(retval);
19.             if(retval.data == 0) continue;
20.             tmp = CreateDFAState(retval);
21.             dfa[dv][i] = tmp;
22.             if(visited.count(tmp) == 0)
23.             {
24.                 visited.push(tmp);
25.                 st.push(tmp);
26.             }
27.         }
28.     }}
```

Kode Sumber 4.6.1 Implementasi fungsi ConvertNFAtoDFA

```
1.  void   MoveClosure(dfa_label<int>   node,   int
2.  symbol, dfa_label<int> &retval)
3.  {
4.      retval.clear();
5.      for( int i=0; i<node.data; i++)
6.      {
7.          for(int                              j=0;
    j<nfa_nodes[node.dt[i]][symbol].size(); j++)
8.      {          retval.push(nfa_nodes[node.dt[i]][sy
9.  mbol][j]);
10.         }
11.     }
12.        retval.calculateHash();
13. }
```

Kode Sumber 4.6.2 Implementasi fungsi MoveClosure

```
1.  void EpsilonClosure(dfa_label<int> &retval)
2.  {
3.      stack<int> nodes;
4.      int tmp;
5.      for( int i=0; i < retval.size();i++)
6.      {
7.          nodes.push(retval.dt[i]);
8.      }
9.      while(nodes.empty() == false)
10.     {
11.         int t = nodes.top();nodes.pop();
12.         for(int  i=0;  i<nfa_nodes[t][2].size();
    i++)
13.         {
14.             tmp = nfa_nodes[t][2][i];
15.             if(retval.count(tmp) == 0)
16.             {
17.                 retval.push(tmp);
18.                 nodes.push(tmp);
19.             }
20.         }
21.     }
22.     retval.calculateHash();
23. }
```

Kode Sumber 4.6.3 Implementasi fungsi EpsilonClosure

Fungsi pendukung selanjutnya adalah fungsi EpsilonClosure. Fungsi ini digunakan untuk melakukan pengecekan terhadap perubahan suatu *state* saat transisi epsilon. Fungsi ini berdasarkan pada *pseudocode* pada **Gambar 3.2.4.** Implementasi fungsi EpsilonClosure dapat dilihat pada **Kode Sumber 4.6.3.**

Fungsi pendukung terakhir adalah fungsi CreateDFAState yang berguna untuk membuat sebuah *state* DFA baru yang kemudian dimasukkan kepada *state machine* DFA. Fungsi ini diimplementasi berdasarkan *pseudocode* pada **Gambar 3.2.6.** Implementasi fungsi dapat dilihat pada **Kode Sumber 4.6.4.**

```
1.  int CreateDFAState(dfa_label<int> label)
2.  {
3.         if(dfa_graph.count(label) == 1)
4.                return dfa_graph[label];
5.
6.      for(int i=0;i<label.data;i++)
7.      {
8.          dfa_lbl[did].push(label.dt[i]);
9.      }
10.     dfa_lbl[did].calculateHash();
11.     dfa_graph[dfa_lbl[did]] = did;
12.     return did++;
13. }
```

Kode Sumber 4.6.4 Implementasi fungsi CreateDFAState

4.7 Implementasi Fungsi MatrixPower

Fungsi MatrixPower merupakan fungsi yang digunakan untuk melakukan perpangkatan matriks. Fungsi ini diimplementasikan dengan berdasar *pseudocode* pada **Gambar 3.2.7** *Pseudocode* fungsi MatrixPower. Implementasi fungsi MatrixPower dapat dilihat pada **Kode Sumber 4.7.1.**

Fungsi MatrixPower memiliki fungsi pendukung yakni fungsi MatrixMultiply. Implementasi fungsi MatrixMultiply dapat dilihat pada **Kode Sumber 4.7.2** dan **Kode Sumber 4.7.3.** Fungsi ini bertugas untuk melakukan perkalian antara dua

matriks persegi berdimensi sama. Dikarenakan *Adjacency Matrix* dari DFA merupakan *Sparse Matrix*, maka perkalian matriks tersebut dioptimasi dengan melakukan pengecekan apakah angka yang mau dikalikan adalah angka 0 atau bukan. Apabila angka yang dikalikan adalah angka 0, maka operasi perkalian dan modulus tidak perlu dilakukan (baris 17).

```
1.  void MatrixPower(long long** adjM, int L,
    long long** result)
2.  {
3.      for(int i=0;i<did;i++)
4.      {
5.          for(int j=0;j<did;j++)
6.          {
7.              result[i][j]=(i==j);
8.          }
9.      }
10.     while(L){
11.         if (L&1)
12.         {
13.             MatrixMultiply(result, adjM);
14.         }
15.         MatrixMultiply(adjM, adjM);
16.         L>>=1;
17.     }
18. }
```

Kode Sumber 4.7.1 Implementasi fungsi MatrixPower

```
1.  void MatrixMultiply(long long** mat1, long
    long** mat2)
2.  {
3.      long long result[did][did];
4.      int i,j,k;
5.      for(i=0;i<did;i++)
6.      {
7.          for(j=0;j<did;j++)
8.          {
9.              result[i][j]=0LL;
10.         }
11.     }
```

Kode Sumber 4.7.2 Implementasi fungsi MatrixMultiply (1)

```
12.     for(i=0;i<did;i++)
13.     {
14.         for(j=0;j<did;j++)
15.         {
16.             for(k=0;k<did;k++)
17.             {
18.                 if(mat1[i][k]    ==    0    ||
    mat2[k][j] == 0 ) continue;

19.     result[i][j]=(result[i][j]+((mat1[i][k])*(mat
    2[k][j])))%MOD_CONSTANT;
20.             }
21.         }
22.     }
23.     for(i=0;i<did;i++)
24.     {
25.         for(j=0;j<did;j++)
26.         {
27.             mat1[i][j]=result[i][j];
28.         }
29.     }
30. }
```

Kode Sumber 4.7.3 Implementasi fungsi MatrixMultiply (2)

4.8 Implementasi Struktur Data Penunjang

Pada implementasi program, untuk mencapai batas waktu berjalan program yang ada pada SPOJ diperlukan beberapa optimasi tambahan yakni dengan mengimplementasikan struktur data *set*. Hal ini dikarenakan untuk melakukan pencarian pada C++ *map* dan C++ *set* sebagai *key* pada *map* tidak cukup cepat untuk melewati batas waktu berjalan program. Pencarian suatu data jika diberikan suatu *key* pada map akan memiliki kompleksitas $O(\log_2 M)$ dimana M adalah banyak data yang disimpan pada *map* dikarenakan implementasi C++ *map* menggunakan struktur data *Binary Search Tree* (BST). Selain itu untuk mencari suatu data dengan C++ *set* sebagai *key* diperlukan melakukan komparasi *set* pada setiap node BST yang memiliki kompleksitas $O(N)$ dimana N adalah bayaknya elemen dalam *set*. Akibatnya, kompleksitas

yang dibutuhkan untuk pencarian sebuah data dalam *map* dengan *set* sebagai *key* adalah $Nxlog_2M$.

Untuk meningkatkan performa pencarian tersebut, dibuatlah implementasi *struct* dengan nama dfa_label yang menggunakan *hash* sebagai komparator. Implementasi *struct* dfa_label dapat dilihat pada **Kode Sumber 4.8.1**, **Kode Sumber 4.8.2**, dan **Kode Sumber 4.8.3**. *Hash* pada dfa_label diperoleh dari perhitungan kombinasi elemen dalam dfa_label yang dilakukan dalam fungsi calculateHash (baris 48 s.d 55). Implementasi *hash* akan membuat komparasi antar *struct* dfa_label memiliki kompleksitas $O(1)$. Implementasi fungsi komparasi dapat dilihat pada baris 16 s.d. 27.

```
1.  template <typename T>
2.  struct dfa_label{
3.      unsigned long long hash_id;
4.      T dt[101];
5.      int data;
6.
7.      dfa_label()
8.      {
9.          hash_id = 0;
10.         data = 0;
11.     }
12.     // struct comparator with '==' operator
13.     bool operator==(const dfa_label &o) const
14.     {
15.         return hash_id == o.hash_id;
16.     }
17.
18.     // struct comparator with '==' operator
19.     // if |data| < cmp.|data| ret true;
20.     // if data[i] < cmp.data[i] ret true;
21.     // else ret false;
22.     bool operator<(const dfa_label &o) const
23.     {
24.             return hash_id < o.hash_id;
25.     }
```

Kode Sumber 4.8.1 Implementasi *struct* dfa_label (1)

```
27.     // insert new element to label
28.     // check for duplicates and sort the
29. element after insert
        // if exists ret 0, else return 1
30.     bool push(T ins)
31.     {
32.         for(int i=0;i<data;i++)
33.         {
34.             if(ins == dt[i])
35.             {
36.                 return false;
37.             }
38.         }
39.         dt[data++] = ins;
40.         sort(dt,dt+data);
41.         return true;
42.     }
43.
44.     void calculateHash()
45.     {
46.         hash_id = 0;
47.         for(int i=0;i<data;i++)
48.         {
49.             hash_id=hash_id*211+dt[i];
50.         }
51.     }
52.
53.     // check element in label
54.     // if exists return 1, else return 0
55.     bool count(T ins)
56.     {
57.         for(int i=0;i<data;i++)
58.         {
59.             if(ins == dt[i])
60.             {
61.                 return true;
62.             }
63.         }
64.         return false;
65.     }
```

Kode Sumber 4.8.2 Implementasi *struct* dfa_label (2)

```
66.
67.      // return how much nodes in label
68.      int size()
69.      {
70.          return data;
71.      }
72.
73.      void clear()
74.      {
75.          hash_id = 0;
76.          data = 0;
77.      }
78.
79.  };
80.
```

Kode Sumber 4.8.3 Implementasi *struct* dfa_label (3)

Struktur data penunjang lainnya adalah *struct* pair_int. Struktur data ini merupakan struktur data yang digunakan untuk mempermudah penyimpanan pasangan dari dua bilangan bulat. Implementasi *struct* pair_int dapat dilihat pada **Kode Sumber 4.8.4.**

```
1.  struct pair_int
2.  {
3.      int a;
4.      int b;
5.      pair_int(){ }
6.      pair_int(int _a, int _b)
7.      {
8.          a = _a;
9.          b = _b;
10.     }
11. };
```

Kode Sumber 4.8.4 Implementasi *struct* pair_int

[Halaman ini sengaja dikosongkan]

BAB V
UJI COBA DAN EVALUASI

Pada bab ini akan dijelaskan hasil uji coba dan evaluasi program yang telah selesai diimplementasi.

5.1 Lingkungan Uji Coba

Lingkungan uji coba yang akan digunakan adalah,

1. Perangkat Keras
   Processor Intel® Core™ i3-2310M CPU @ 2.10 GHz
   RAM 4 GB
   Sistem Operasi 64-bit
2. Perangkat Lunak
   Sistem Operasi Windows 8.0
   Integrated Development Environment Code::Blocks 13.12

5.2 Skenario Uji Coba

Ada dua skenario uji coba yang akan dilakukan pada subbab ini. Dua skenario tersebut adalah uji coba kebenaran program dan uji coba kinerja program.

5.2.1 Uji Coba Kebenaran

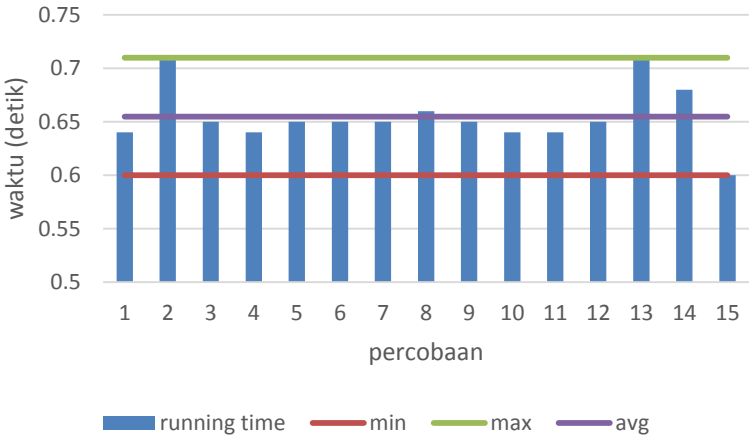
Uji coba kebenaran merupakan uji coba yang dilakukan untuk menguji kebenaran implementasi metode yang digunakan pada program. Uji coba dilakukan dengan mengirim kode sumber pada situs SPOJ dengan kode permasalahan CTSTRING yang diangkat sebagai studi kasus Tugas Akhir ini. Setelah kode sumber dikirim, SPOJ akan memberikan umpan balik terhadap hasil berjalannya program. Hasil yang didapatkan dari SPOJ adalah *accepted*, waktu yang diperlukan untuk penyelesaian permasalahan adalah 0.60 detik dan memori yang dibutuhkan adalah 3MB seperti yang ditunjukkan pada **Gambar 5.2.1**. Hal tersebut menunjukkan bahwa implementasi yang dilakukan berhasil menginterpretasikan *regular*

expression dan mengubahnya ke bentuk DFA untuk melakukan pengecekan *string* yang dapat diterima.

Gambar 5.2.1 Umpan balik dari situs SPOJ

Kemudian dilakukan uji coba dengan mengirim kode sumber sebanyak 15 kali pada permasalahan yang sama untuk melihat variasi hasil yang dihasilkan oleh program. Hasil uji coba tersebut dapat dilihat pada **Gambar 5.2.2**, **Gambar 6.2.1** dan **Tabel 6.2.1.**

Dari hasil uji coba yang dilakukan waktu yang dibutuhkan bervariasi antara 0.6 s.d. 0.71 dengan rata-rata 0.65 detik namun memori yang dibutuhkan program tetap 3 MB.



Gambar 5.2.2 Grafik hasil uji coba pada SPOJ sebanyak 15 kali

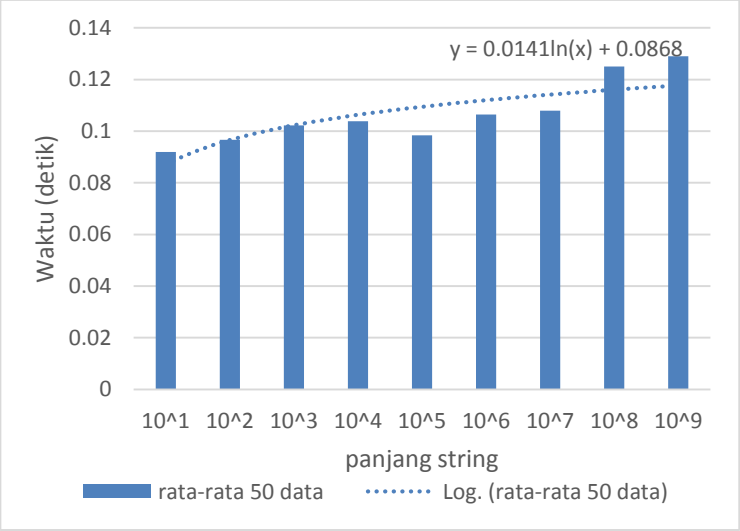
5.2.2 Uji Coba Kinerja

Uji coba kinerja adalah skenario uji coba yang dilakukan dengan parameter tertentu untuk melihat bagaimana performa dari program yang dibuat. Pada uji coba ini parameter yang digunakan adalah panjang *regular expression* yang harus

diproses dan L di mana program harus menghitung berapa banyak *string* yang bisa diterima *regular expression* apabila panjang *string* adalah L.

5.2.2.1 Pengaruh Panjang L Terhadap Waktu

Uji coba yang pertama kali dilakukan adalah dengan memberikan program beberapa panjang L yang berbeda yakni bilangan $10^1, 10^2, 10^3, 10^4 \ldots 10^9$. Panjang *regular expression* yang digunakan adalah 100 karakter. Pada uji coba yang dilakukan, setiap bilangan L yang berbeda akan diuji dengan *regular* expression yang sama sebanyak 50 kali kemudian di rata-rata untuk memperoleh waktu rata-rata untuk memproses *regular expression* dengan panjang 100 dan *string* dengan panjang L.



Gambar 5.2.3 Grafik rata-rata hasil uji coba untuk L yang berbeda

Hasil uji coba tersebut dapat dilihat di **Gambar 5.2.3**. Uji coba ini dimaksudkan untuk melakukan tes performa ketika *regular expression* dicocokan dengan sebuah *string*. Performa

yang diharapkan dalam uji coba ini adalah logaritmik. Untuk *string* yang semakin panjang maka waktu yang dibutuhkan akan semakin lama.

Pada **Gambar 5.2.3** dapat dilihat bahwa dengan perbedaan L yang diberikan, semakin besar L maka semakin besar pula waktu yang diperlukan untuk mengeksekusi program, dan perkembangan waktu yang dihasilkan memenuhi waktu logaritmik seperti yang ditujukkan pada gambar. hal dikarenakan algoritma perkalian matriks yang digunakan menggunakan algoritma dengan kompleksitas $O(\log(L)\, xN^2)$ dimana L adalah besar pangkat dan N adalah dimensi matriks.

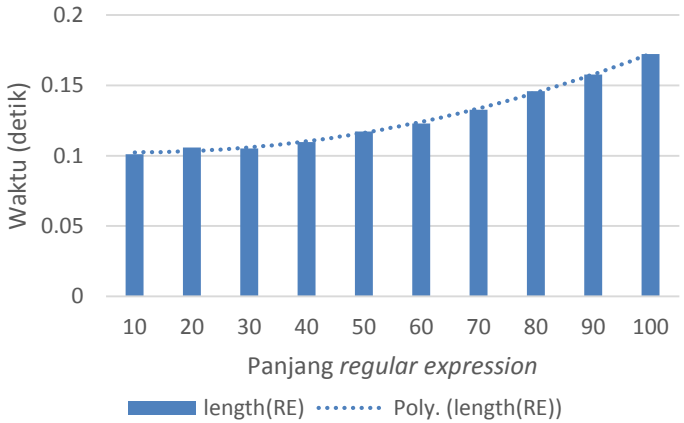
Dari hasil uji coba yang dilakukan dapat dilihat bahwa besar atau kecilnya L memiliki pengaruh terhadap waktu yang diperlukan program untuk melakukan evaluasi permasalahan dengan kompleksitas logaritmik.

5.2.2.2Pengaruh Panjang Regular Expression Terhadap Waktu

Uji coba yang dilakukan selanjutnya adalah dengan menggunakan *regular expression* acak dengan operator yang digunakan hanya operator *concatenate* yang dibangkitkan dengan sebuah program pembangkit uji kinerja. Operator *concatenate* digunakan karena memberikan hasil DFA yang lebih konsisten untuk setiap input dengan panjang *regular expression* yang berbeda, di mana semakin panjang *regular expression* maka *state* DFA yang dihasilkan juga akan semakin banyak pula. Parameter yang digunakan pada uji coba adalah panjang *regular expression*. Panjang yang diujikan adalah 10, 20, 30, 40, 50, 60, 70, 80, 90, dan 100 karakter. Bilangan L yang digunakan pada uji coba ini adalah 10^9.

Data hasil uji coba tersebut dapat dilihat pada **Gambar 5.2.4**. Uji coba ini dimaksudkan untuk melakukan tes performa konversi *regular expression* menjadi NFA dan kemudian DFA. Performa algoritma yang diharapkan adalah polinomial berderajat 2, Pada **Gambar 5.2.4** terlihat jika *regular expression* yang perlu dievaluasi semakin panjang makan semakin besar

pula waktu yang diperlukan untuk melakukan konversi terhadap *regular expression* tersebut dan waktu berjalan program memenuhi kurva polinomial berderajat 2, hal ini dikarenakan saat pembentukan NFA operasi yang diperlukan adalah sebanyak $O(2xN)$ atau bisa dibilang linear untuk setiap operator *concatenate* yang dilakukan, sedangkan untuk konstruksi NFA ke DFA diperlukan operasi sebanyak $O(M^2)$ dimana M adalah jumlah *state* NFA.



Gambar 5.2.4 Grafik uji coba dengan panjang RE yang berbeda

Dari hasil tersebut dapat dilihat bahwa semakin panjang *regular expression* yang diberikan, maka semakin besar pula waktu yang dibutuhkan program untuk melakukan konversi *regular expression* tersebut.

[Halaman ini sengaja dikosongkan]

BAB VI
KESIMPULAN DAN SARAN

Pada bab ini akan dijelaskan mengenai kesimpulan dari proses dan uji coba dari program dan saran untuk pengembangan dari program itu sendiri.

6.1 Kesimpulan

Dari hasil uji coba program yang telah dilakukan dengan implementasi model NFA dan DFA untuk menginterpretasikan *regular expression* dapat ditarik beberapa kesimpulan sebagai berikut,

1. Implementasi yang dilakukan berhasil dengan benar menyelesaikan permasalahan studi kasus yang digunakan dalam penyusunan tugas akhir, yakni studi kasus SPOJ klasik 10354 dengan rata-rata waktu berjalan program sebesar 0.65 detik dan memori 3MB.
2. Untuk evaluasi *regular expression*, ditandai dengan besarnya L yang berbeda memiliki pengaruh terhadap waktu yang dibutuhkan program secara logaritmik yakni sebesar $O(M^2 \log(L))$, dimana M adalah dimensi *Adjacency Matrix* dari DFA.
3. Untuk konversi *regular expression* menjadi *state machine* NFA dan DFA, panjang *regular expression* memiliki pengaruh terhadap waktu yang dibutuhkan secara polinomial yakni $O(2N) + (2^N)$, $O(2N)$ untuk konversi *regular expression* menjadi NFA, dan $O(2^N)$ untuk konversi NFA menjadi DFA. Kompleksitas tersebut memiliki makna jika semakin panjang *regular expression* maka semakin besar pula waktu yang dibutuhkan.

6.2 Saran

Saran yang dapat digunakan untuk melakukan pengembangan program interpretasi regular expression menjadi

model *state machine* NFA dan DFA yakni menggunakan metode Glushkov untuk interpretasi *regular expression*, metode Glushkov memiliki properti khusus pada NFA yang terbentuk, yakni NFA tersebut tidak memiliki transisi epsilon. NFA hasil dari metode Glushkov sudah dapat digunakan untuk melakukan pencarian *string* dengan cepat tanpa perlu mengubah NFA ke DFA.

DAFTAR ACUAN

[1] G. Navaro and M. Rafinot, Flexible Pattern Matching in Strings, Cambridge: Cambridge University Press, 2002.

[2] J. Power, "Constructing a DFA from an NFA (Subset Construction)," National University of Ireland, 29 November 2002. [Online]. Available: http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node9.html. [Accessed 25 April 2015].

[3] kuruma, "A tutorial on Fast Modulo Multiplication (Exponential Squaring)," 12 Agustus 2013. [Online]. Available: http://discuss.codechef.com/questions/20451/a-tutorial-on-fast-modulo-multiplication-exponential-squaring. [Accessed 25 April 2015].

[4] V. Jalan, "Count Strings," 9 September 2012. [Online]. Available: http://www.spoj.com/problems/CTSTRING/. [Accessed 10 Desember 2014].

[Halaman ini sengaja dikosongkan]

LAMPIRAN A

Tabel 6.2.1 Hasil uji coba 15 kali pada permasalahan CTSTRING

No.	Hasil	Waktu (detik)	Memori (MB)
1	Accepted	0.64	3
2	Accepted	0.71	3
3	Accepted	0.65	3
4	Accepted	0.64	3
5	Accepted	0.65	3
6	Accepted	0.65	3
7	Accepted	0.65	3
8	Accepted	0.66	3
9	Accepted	0.65	3
10	Accepted	0.64	3
11	Accepted	0.64	3
12	Accepted	0.65	3
13	Accepted	0.71	3
14	Accepted	0.68	3
15	Accepted	0.6	3

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
14379221		2015-06-02 19:40:56	Count Strings	accepted edit ideone it	0.64	3.0M	C++ 4.3.2
14379213		2015-06-02 19:40:18	Count Strings	accepted edit ideone it	0.71	3.0M	C++ 4.3.2
14379208		2015-06-02 19:39:38	Count Strings	accepted edit ideone it	0.65	3.0M	C++ 4.3.2
14379199		2015-06-02 19:38:58	Count Strings	accepted edit ideone it	0.64	3.0M	C++ 4.3.2
14379194		2015-06-02 19:38:24	Count Strings	accepted edit ideone it	0.65	3.0M	C++ 4.3.2
14379186		2015-06-02 19:37:36	Count Strings	accepted edit ideone it	0.65	3.0M	C++ 4.3.2
14379183		2015-06-02 19:37:04	Count Strings	accepted edit ideone it	0.65	3.0M	C++ 4.3.2
14379178		2015-06-02 19:36:26	Count Strings	accepted edit ideone it	0.66	3.0M	C++ 4.3.2
14379175		2015-06-02 19:35:57	Count Strings	accepted edit ideone it	0.65	3.0M	C++ 4.3.2
14379170		2015-06-02 19:35:28	Count Strings	accepted edit ideone it	0.64	3.0M	C++ 4.3.2
14379167		2015-06-02 19:35:07	Count Strings	accepted edit ideone it	0.64	3.0M	C++ 4.3.2
14379165		2015-06-02 19:34:42	Count Strings	accepted edit ideone it	0.65	3.0M	C++ 4.3.2
14379155		2015-06-02 19:33:45	Count Strings	accepted edit ideone it	0.71	3.0M	C++ 4.3.2
14374047		2015-06-02 06:58:55	Count Strings	accepted edit ideone it	0.68	3.0M	C++ 4.3.2
14373986		2015-06-02 06:46:05	Count Strings	accepted edit ideone it	0.60	3.0M	C++ 4.3.2

Gambar 6.2.1 Hasil uji coba 15 kali pada permasalahan CTSTRING

BIODATA PENULIS

Penulis memiliki nama lengkap Muhammad Yunus Bahari, lahir di Blitar pada 26 Mei 1993. Anak pertama dari 4 bersaudara. Penulis telah menempuh pendidikan formal pada jenjang TK sampai dengan S-1 di TK Aisyah Bustanul Athfal kab. Sumbawa Besar (1997-1999), SDN Kalipang 1 kab. Blitar (1999-2002), SDN Tlogo 1 kab. Blitar (2002-2005), SMPN 3 Blitar (2005-2008), SMAN 1 Blitar (2008-2011), Teknik Informatika Institut Teknologi Sepuluh Nopember Surabaya (2011-2015). Penulis pernah menjuarai kompetisi SNITCH I FTIf ITS dalam kategori pengembangan permainan. Semenjak tahun 2012 penulis telah aktif sebagai anggota dalam Himpunan Mahasiswa Teknik Computer-Informatika (HMTC) ITS. Penulis pernah menjadi anggota (2012) dan ketua sub-divisi (2013) dalam National Logic Competition (NLC) Schematics ITS. Penulis juga pernah menjadi Koordinator Lab Algoritma dan Pemrograman (2014-2015) dan menjadi asisten mata kuliah Pemrograman Terstruktur (2012), Basis Data (2013), Pemrograman Framework .NET (2013) di Teknik Informatika ITS dan Basis Data PIKTI ITS (2013).