taW

;/--n

**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]***

**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**

# F:\Dropbox\Dropbox\Semester 8\TA\mine\0098.jpg

# 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)

1. 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



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

[LEMBAR PENGESAHAN v](#_Toc425380718)

[*Abstrak* vii](#_Toc425380719)

[*Abstract* ix](#_Toc425380720)

[KATA PENGANTAR xi](#_Toc425380721)

[DAFTAR ISI xiii](#_Toc425380722)

[DAFTAR GAMBAR xv](#_Toc425380723)

[DAFTAR TABEL xvii](#_Toc425380724)

[DAFTAR KODE SUMBER xix](#_Toc425380725)

[BAB I PENDAHULUAN 1](#_Toc425380726)

[1.1 Latar Belakang 1](#_Toc425380727)

[1.2 Rumusan Masalah 1](#_Toc425380728)

[1.3 Batasan Masalah 2](#_Toc425380729)

[1.4 Tujuan 2](#_Toc425380730)

[1.5 Manfaat 2](#_Toc425380731)

[1.6 Metodologi 3](#_Toc425380732)

[BAB II TINJAUAN PUSTAKA 5](#_Toc425380733)

[2.1 Regular Expression 5](#_Toc425380734)

[2.2 String Matching Menggunakan Regular Expression 6](#_Toc425380735)

[2.3 Nondeterministic Finite Automaton 7](#_Toc425380736)

[2.4 Metode Thompson 9](#_Toc425380737)

[2.5 Deterministic Finite Automaton 11](#_Toc425380738)

[2.6 Metode Subset Construction 11](#_Toc425380739)

[2.7 Perpangkatan Matriks 13](#_Toc425380740)

[2.8 Permasalahan CTSTRING di SPOJ 14](#_Toc425380741)

[2.9 Desain Umum Penyelesaian Permasalahan CTSTRING 16](#_Toc425380742)

[2.10 Ilustrasi Penyelesaian permasalahan CTSTRING 17](#_Toc425380743)

[BAB III DESAIN PERANGKAT LUNAK 23](#_Toc425380744)

[3.1 Desain Sistem Secara Umum 23](#_Toc425380745)

[3.2 Desain Algoritma 24](#_Toc425380746)

[3.2.1 Desain fungsi Preprocess 24](#_Toc425380747)

[3.2.2 Desain fungsi ConvertREtoNFA 25](#_Toc425380748)

[3.2.3 Desain fungsi ConvertNFAtoDFA 27](#_Toc425380749)

[3.2.4 Desain fungsi MatrixPower 30](#_Toc425380750)

[3.3 Desain Struktur Data 31](#_Toc425380751)

[3.4 Desain Pembangkit Kasus untuk Uji Kinerja. 32](#_Toc425380752)

[BAB IV IMPLEMENTASI 35](#_Toc425380753)

[4.1 Lingkungan Implementasi 35](#_Toc425380754)

[4.2 Konstanta dan Variabel Global 35](#_Toc425380755)

[4.3 Implementasi Fungsi Main 36](#_Toc425380756)

[4.4 Implementasi Fungsi Preprocess 39](#_Toc425380757)

[4.5 Implementasi Fungsi ConvertREtoNFA 39](#_Toc425380758)

[4.6 Implementasi Fungsi ConvertNFAtoDFA 41](#_Toc425380759)

[4.7 Implementasi Fungsi MatrixPower 44](#_Toc425380760)

[4.8 Implementasi Struktur Data Penunjang 46](#_Toc425380761)

[BAB V UJI COBA DAN EVALUASI 51](#_Toc425380762)

[5.1 Lingkungan Uji Coba 51](#_Toc425380763)

[5.2 Skenario Uji Coba 51](#_Toc425380764)

[5.2.1 Uji Coba Kebenaran 51](#_Toc425380765)

[5.2.2 Uji Coba Kinerja 52](#_Toc425380766)

[BAB VI KESIMPULAN DAN SARAN 57](#_Toc425380767)

[6.1 Kesimpulan 57](#_Toc425380768)

[6.2 Saran 57](#_Toc425380769)

[DAFTAR ACUAN 59](#_Toc425380770)

[LAMPIRAN A 61](#_Toc425380771)

[BIODATA PENULIS 63](#_Toc425380772)

# DAFTAR GAMBAR

[Gambar 2.3.1 Representasi NFA dalam *directed* *graph* 8](#_Toc425380773)

[Gambar 2.4.1 *Automaton* transisi epsilon 9](#_Toc425380774)

[Gambar 2.4.2 *Automaton* dengan sebuah masukan karakter 9](#_Toc425380775)

[Gambar 2.4.3 *Automaton* operator *concatenate* 10](#_Toc425380776)

[Gambar 2.4.4 *Automaton* operator Union 10](#_Toc425380777)

[Gambar 2.4.5 *Automaton* operator Klenee star 10](#_Toc425380778)

[Gambar 2.4.6 *Automaton* awal yang terbentuk 11](#_Toc425380779)

[Gambar 2.4.7 *Automaton* setelah operator *Klenee star* 11](#_Toc425380780)

[Gambar 2.4.8 *Automaton* setelah operator *Union* 11](#_Toc425380781)

[Gambar 2.4.9 NFA representasi dari *regular expression*  11](#_Toc425380782)

[Gambar 2.8.1 Deskripsi permasalahan CTSTRING 14](#_Toc425380783)

[Gambar 2.8.2 Contoh masukan dan keluaran permasalahan 15](#_Toc425380784)

[Gambar 2.9.1 Diagram Alur penyelesaian permasalahan 16](#_Toc425380785)

[Gambar 2.10.1 Evaluasi operasi (a\*) pertama dan kedua 18](#_Toc425380786)

[Gambar 2.10.2 Evaluasi operasi (b+(a\*)) 18](#_Toc425380787)

[Gambar 2.10.3 Evaluasi operasi ((a\*)+(b+(a\*))) 18](#_Toc425380788)

[Gambar 2.10.4 hasil konversi NFA ke DFA 19](#_Toc425380789)

[Gambar 2.10.5 DFA yang telah disederhanakan 19](#_Toc425380790)

[Gambar 2.10.6 *Adjacency matrix* dari DFA yang dibuat 20](#_Toc425380791)

[Gambar 2.10.7 *Adjacency matrix* untuk perpindahan sebanyak 2 langkah 20](#_Toc425380792)

[Gambar 2.10.8 *Adjacency matrix* setelah dipangkatkan 100 21](#_Toc425380793)

[Gambar 3.1.1 *Pseudocode* fungsi main 23](#_Toc425380794)

[Gambar 3.2.1 *Pseudocode* fungsi preprocess 24](#_Toc425380795)

[Gambar 3.2.2 *Pseudocode* fungsi ConvertREtoNFA 26](#_Toc425380796)

[Gambar 3.2.3 *Pseudocode* fungsi ConvertNFAtoDFA 27](#_Toc425380797)

[Gambar 3.2.4 *Pseudocode* fungsi EpsilonClosure 29](#_Toc425380798)

[Gambar 3.2.5 *Pseudocode* fungsi MoveClosure 29](#_Toc425380799)

[Gambar 3.2.6 *Pseudocode* fungsi CreateDFAState 30](#_Toc425380800)

[Gambar 3.2.7 *Pseudocode* fungsi MatrixPower 30](#_Toc425380801)

[Gambar 3.4.1 *Pseudocode* fungsi utama pembangkit kasus uji 32](#_Toc425380802)

[Gambar 3.4.2 *Pseudocode* fungsi Expand 33](#_Toc425380803)

[Gambar 5.2.1 Umpan balik dari situs SPOJ 52](#_Toc425380804)

[Gambar 5.2.2 Grafik hasil uji coba pada SPOJ sebanyak 15 kali 52](#_Toc425380805)

[Gambar 5.2.3 Grafik rata-rata hasil uji coba untuk L yang berbeda 53](#_Toc425380806)

[Gambar 5.2.4 Grafik uji coba dengan panjang RE yang berbeda 55](#_Toc425380807)

[Gambar 6.2.1 Hasil uji coba 15 kali pada permasalahan CTSTRING 62](#_Toc425380808)

# DAFTAR TABEL

[Tabel 2.3.1 Representasi NFA dalam tabel 8](#_Toc425380809)

[Tabel 2.10.1 Iterasi dalam konversi NFA menjadi DFA 18](#_Toc425380810)

[Tabel 6.2.1 Hasil uji coba 15 kali pada permasalahan CTSTRING 61](#_Toc425380811)

***[Halaman ini sengaja dikosongkan]***

# DAFTAR KODE SUMBER

[Kode Sumber 4.2.1 Konstanta dan variabel global 36](#_Toc425380812)

[Kode Sumber 4.3.1 Implementasi fungsi main(1) 36](#_Toc425380813)

[Kode Sumber 4.3.2 Implementasi fungsi main(2) 37](#_Toc425380814)

[Kode Sumber 4.3.3 Implementasi fungsi main(3) 38](#_Toc425380815)

[Kode Sumber 4.3.4 Implementasi fungsi Initialize 38](#_Toc425380816)

[Kode Sumber 4.4.1 Implementasi fungsi Preprocess 39](#_Toc425380817)

[Kode Sumber 4.5.1 Implementasi fungsi ConvertREtoNFA (1) 39](#_Toc425380818)

[Kode Sumber 4.5.2 Implementasi fungsi ConvertREtoNFA (2) 40](#_Toc425380819)

[Kode Sumber 4.5.3 Implementasi fungsi ConvertREtoNFA (3) 41](#_Toc425380820)

[Kode Sumber 4.6.1 Implementasi fungsi ConvertNFAtoDFA 42](#_Toc425380821)

[Kode Sumber 4.6.2 Implementasi fungsi MoveClosure 43](#_Toc425380822)

[Kode Sumber 4.6.3 Implementasi fungsi EpsilonClosure 43](#_Toc425380823)

[Kode Sumber 4.6.4 Implementasi fungsi CreateDFAState 44](#_Toc425380824)

[Kode Sumber 4.7.1 Implementasi fungsi MatrixPower 45](#_Toc425380825)

[Kode Sumber 4.7.2 Implementasi fungsi MatrixMultiply (1) 45](#_Toc425380826)

[Kode Sumber 4.7.3 Implementasi fungsi MatrixMultiply (2) 46](#_Toc425380827)

[Kode Sumber 4.8.1 Implementasi *struct* dfa\_label (1) 47](#_Toc425380828)

[Kode Sumber 4.8.2 Implementasi *struct* dfa\_label (2) 48](#_Toc425380829)

[Kode Sumber 4.8.3 Implementasi *struct* dfa\_label (3) 49](#_Toc425380830)

[Kode Sumber 4.8.4 Implementasi *struct* pair\_int 49](#_Toc425380831)

***[Halaman ini sengaja dikosongkan]***

# PENDAHULUAN

## 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*.

## 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?

## 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 10­9.

## 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.

## Manfaat

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

## 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.

1. 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.

1. 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++.

1. Pengujian dan evaluasi

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

1. 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]***

# 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.

## Regular Expression

Sebuah *string* didefinisikan sebagai rangkaian karakter berhingga yang ada dalam semesta simbol . Sebagai contoh FAZZIBASHASOL adalah *string*  dalam semesta simbol [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*  adalah sebuah *string* yang memiliki simbol dalam set, yang didefinisikan secara rekursifsebagai *string* kosong *,* simboldi mana *,* dan *,, ,* dan *,* di manadanadalah *regular expression* [1].

Salah satu contoh *regular expression* adalah . Untuk kemudahan membaca *regular expression* sering kali ditulis menjadi . Sehingga contoh *regular expression* tersebut dapat disederhanakan menjadi . 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 dimana yang memiliki struktur *RE* sebagai berikut [1],

* Jika adalah , maka = {}, *string* kosong.
* Jika adalah , maka = , sebuah *string* dengan panjang satu karakter.
* Jika adalah *regular expression* dengan bentuk , maka .
* Jika adalah *regular expression* dengan bentuk , maka . di mana adalah kumpulan string w sedemikian hingga , dimana dan . Operator “” adalah operator *concatenate*.
* Jika adalah *regular expression* dengan bentuk , maka . Operator “|” adalah operator *Union*.
* Jika adalah *regular expression* dengan bentuk , maka . di mana dan untuk semua . Operator “\*” adalah operator *Klenee Star*.

Sebagai contoh *regular expression* akan merepresentasikan bahasa

.

## 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 diberikansecara 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.

## Nondeterministic Finite Automaton

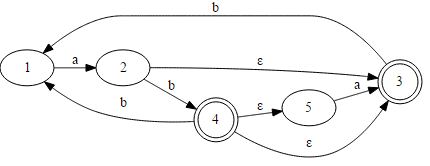
*Nondeterministic Finite Automaton* adalah sebuah *state machine* yang memiliki sekumpulan *state* berhingga *Q*, sekumpulan simbol sebagai masukan transisi *state*, fungsi transisi *F* dari *state* dengan masukan simbol menuju *state* , sekumpulan state sebagai *state* NFA selesai, dan sebuah *state* 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 . *state* awal NFA dimulai dari *state*  dan memiliki dua *state* selesai yakni . 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 sebagai simbol masukan dalam **Tabel 2.3.1**.

Tabel 2.3.1 Representasi NFA dalam tabel

|  |  |  |  |
| --- | --- | --- | --- |
|  | a | b |  |
|  | 2 | - | - |
|  | - | 4 | 3 |
|  | - | 1 | - |
|  | - | 1 | 3,5 |
|  | 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 .



Gambar 2.3.1 Representasi NFA dalam *directed* *graph*

Hal ini membuat NFA menjadi sangat lambat ketika digunakan untuk melakukan pencarian *string*, terlebihjika 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.

## 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  *states* dan maksimal transisi dimana 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 ke 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 ke terjadi apabila karakter dimasukkan sebagai simbol masukan ke NFA saat sebagai *state* aktif.



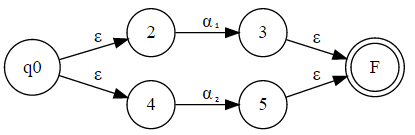
Gambar 2.4.2 *Automaton* dengan sebuah masukan karakter

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



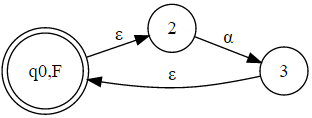
Gambar 2.4.3 *Automaton* operator *concatenate*

* *Automaton* untuk operator “|” (*Union*) ditunjukkan pada **Gambar 2.4.4**. *Automaton* ini berarti perpindahan *state* dari ke dapat terjadi apabila masukan yang diberikan merupakan salah satu dari simbol atau.



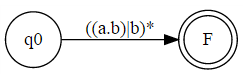
Gambar 2.4.4 *Automaton* operator Union

* *Automaton* untuk operator “\*” (*Klenee* *star*) ditunjukkan pada **Gambar 2.4.5**. *Automaton* berarti perpindahan *state* dari ke dapat terjadi dengan tanpa masukan karena adalah *state* yang sama dengan atau dengan masukan simbol .

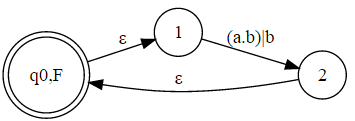


Gambar 2.4.5 *Automaton* operator Klenee star

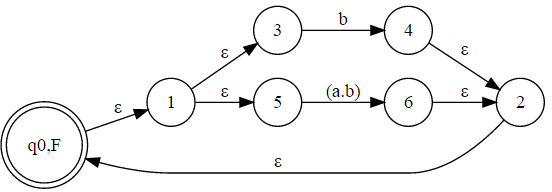
Dengan metode di atas contoh *regular expression*  akan terbentuk NFA dengan melakukan operasi yang sudah didefinisikan oleh metode tersebut dengan proses yang ditunjukan secara berurutan pada **Gambar 2.4.6, Gambar 2.4.7, Gambar 2.4.8,** dan **Gambar 2.4.9**.



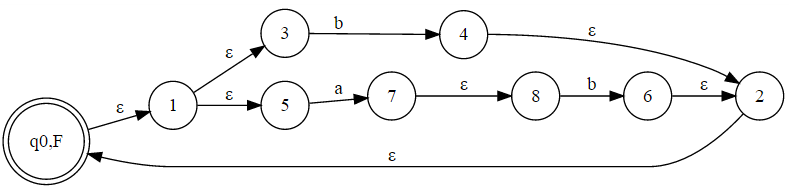
Gambar 2.4.6 *Automaton* awal yang terbentuk



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*

## 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.

## 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 .
   1. 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.
   2. 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].

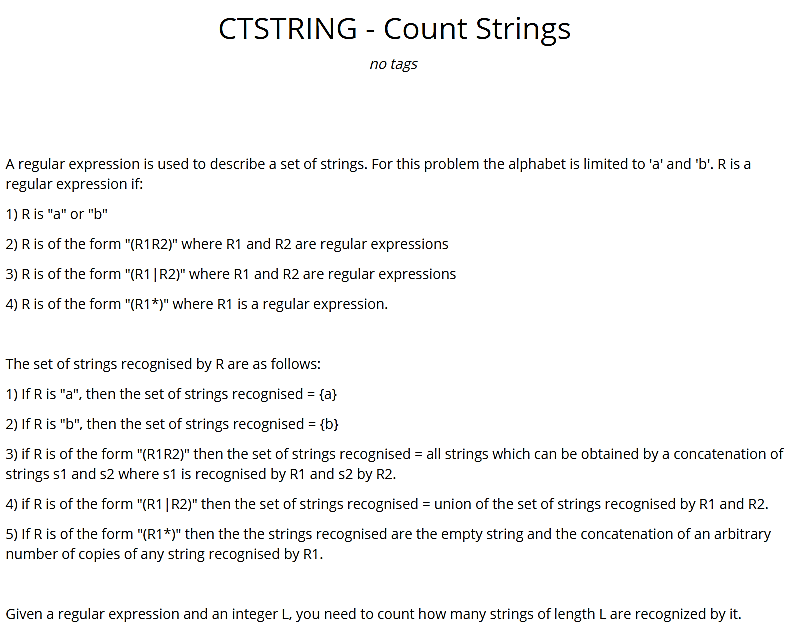
## Perpangkatan Matriks

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 . 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 . Teknik ini secara rekursif didefinisikan pada persamaan 1 [3].

## Permasalahan CTSTRING di SPOJ



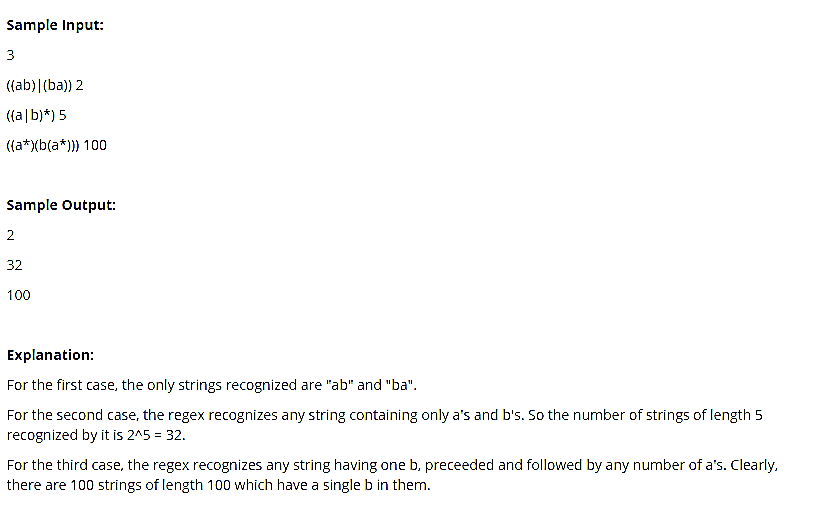
Gambar 2.8.1 Deskripsi permasalahan CTSTRING

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* REdan sebuah bilangan bulat L,dengan batasan *regular expression* memiliki tiga operator yakni *concatenate*, *union,* dan *klenee star* dan simbol . 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 ). Contoh masukan dan keluaran juga dapat dilihat pada **Gambar 2.8.2**.

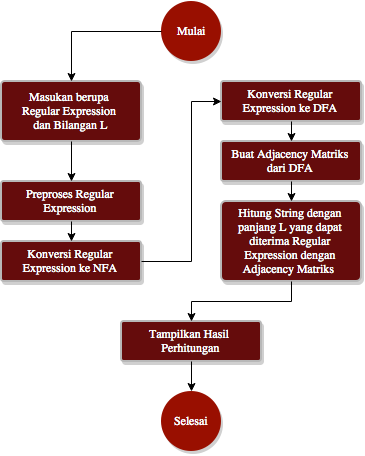


Gambar 2.8.2 Contoh masukan dan keluaran permasalahan

Adapun batasan dari permasalahan tersebut adalah,

1. .
2. .
3. .
4. Dinilai dalam lingkungan penilaian Intel G860 3Ghz
5. Batasan waktu 0.1 detik – 0.150 detik.
6. Batasan memori 1536 MB.

## 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.

## Ilustrasi Penyelesaian permasalahan CTSTRING

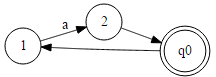
Pada permasalahan CTSTRING diberikan dua masukan yaitu *regular expression RE* dan bilangan *L*. Contoh yang akan dibahas diasumsikan dengan masukan sebagai berikut, dan .

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 .

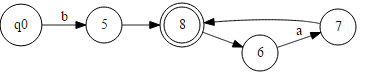
Proses konversi *regular expression* ke NFA dilakukan setelah preproses. Proses ini dilakukan dengan melakukan iterasi setiap karakter *RE* hasil preproses. Pada proses iterasi akan dibentuk kombinasi *automaton* untuk setiap evaluasi operator. Pada contoh di atas akan ada 4 evaluasi operator yang akan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

Evaluasi pertama pertama dan kedua adalah , ketiga adalah , dan terakhir . 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**.

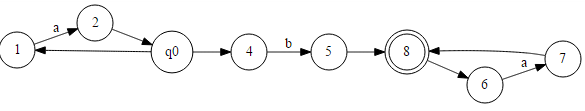
Hasil akhir dari tahap tersebut (**Gambar 2.10.3**) merupakan NFA yang kemudian digunakan untuk konversi menjadi DFA. Proses konversi DFA dilakukan dengan proses yang telah dijelaskan pada subbab **2.6**. Inisialisasi awal DFA dengan melakukan *EpsilonClosure* dari awal DFA yakni *state* 9.

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**.

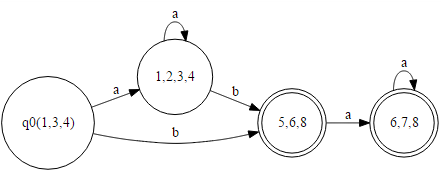
Tabel 2.10.1 Iterasi dalam konversi NFA menjadi DFA

|  |  |  |  |
| --- | --- | --- | --- |
| 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 | - |

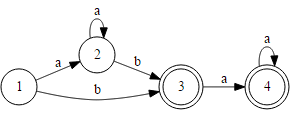
Selesai iterasi untuk membangkitkan DFA dari NFA selesai maka DFA yang terbentuk dapat dilihat pada **Gambar 2.10.4**, dan **Gambar 2.10.5**.

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* 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 percobaan untuk menguji semua *string* dengan panjang 4. Namun pada permasalahan studi kasus yang digunakan, batas atas *L* adalah dan tentu saja untuk mencoba seluruh kemungkinan *string* dengan panjang tersebut sangat tidak mungkin karena akan diperlukan 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**.



Gambar 2.10.6 *Adjacency matrix* dari DFA yang dibuat

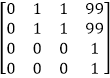
Dari matriks tersebut dapat dilihat relasi antar *state* sebagai contoh pada baris pertama {0,1,1,0} yang memiliki arti *state* 1 memiliki hubungan ke *state* 2, dan 3 dengan satu langkah perpindahan, untuk dua langkah perpindahan maka *adjacency matrix* yang dihasilkan dapat dilihat pada **Gambar 2.10.7**.



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 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.



Gambar 2.10.8 *Adjacency matrix* setelah dipangkatkan 100

Jadi dapat disimpulkan, dengan *regular expression* dan ada 100 *string* dengan panjang 100 yang diterima oleh *regular expression RE*.

***[Halaman ini sengaja dikosongkan]***

# DESAIN PERANGKAT LUNAK

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

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

## 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.

### 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.  2.  3.  4, | for i=0 to length(RE)  if (RE[i] is alphabet or ‘(’) and (RE[i-1] is alphabet or ‘)’)  insert operator “.” to RE[i+1]  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.

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

### 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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | DFA := create empty state machine  NFAinitialState = NFA.getInitialstate()  St := EpsilonClosure(NFAinitialState)  DFASt := CreateDFAState(St, &DFA)  push DFASt to stack tmp  while tmp is not empty  pop tmp and assign to dv  for i=0 to sizeof(alphabet)  St := MoveClosure(dv,alphabet[i])  St := EpsilonClosure(St)  if St is empty set  continue loop  else  DFASt := CreateDFAState(St, &DFA)  Assign alphabet[i] as transition symbol from dv to DFASt  If DFAst not already created before  push DFAst to stack tmp  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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | for i=0 to sizeof(St)  add st to retSt  push st to stack tmp  while stack tmp is not empty  pop tmp and assign to t  r:=next state that can be reached from t with epsilon transition  for i=0 to r  if t.next[i] not exist in retSt  add t.next[i] to retSt  push t.next[i] to stack tmp  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.  2.  3.  4. | for i=0 to sizeof(St)  if St.next(alphabet) is not empty  add St.next(alphabet) to retSt  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.  2.  3.  4. | if St already created and exists in DFA  return St  else add St to DFA  return St |

Gambar 3.2.6 *Pseudocode* fungsi CreateDFAState

### Desain fungsi MatrixPower

|  |  |
| --- | --- |
| \* AdjM adjacency matrix  \* L  power  \* retAdjM Lth power Matrix  retAdjM = MatrixPOwer(AdjM, L) | |
| 1.  3.  4.  5.  6.  7.  8. | Create identity matrix I that has same dimension as adjM  while L>0  if L is odd  I := AdjM x I  AdjM := AdjM x AdjM  L := L/2  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.

## 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*.

## 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.  2.  3.  4.  5.  6.  7.  8. | Initialize random seed()  print(number of test case)  for i=0 to number of test case  initialize str = “a”;  while length of str < 100  index = rand\_int() between [1…length(str)]  str = Expand(str)  print(str+” ”+rand\_int() between 1…) |

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20 | if strIN[index] is “a” or “b” change index to index of not “a” or “b” character  str = strIN  I = rand\_int() with range [1…3]  J = rand\_int() with range [1…2]  if I = 1  if J = 1  insert “(ab)” to str[index+1]  else  insert “(ba)” to str[index+1]  else if I = 2  if J = 1  insert “(a|b)” to str[index+1]  else  insert “(b|a)” to str[index+1]  else  if J = 1  insert (a\*) to str[index+1]  else  insert (b\*) to str[index+1]  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…109].

***[Halaman ini sengaja dikosongkan]***

# 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++.

## 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

* 1. Perangkat Lunak

Sistem Operasi Windows 8.0

Integrated Development Environment Code::Blocks 13.12

## 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.  2.  3.  4.  5.  6.  7.  8.  9.  10. | #define MOD\_CONSTANT 1000000007  #define MAX\_NODE 200  stack<pair\_int> states;  vector<int> nfa\_nodes[MAX\_NODE][3];  map<dfa\_label<int>, int> dfa\_graph;  int dfa[MAX\_NODE][2];  dfa\_label<int> dfa\_lbl[MAX\_NODE];  int nid = 0;  int did = 0; |

Kode Sumber 4.2.1 Konstanta dan variabel global

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

Kode Sumber 4.3.1 Implementasi fungsi main(1)

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

Kode Sumber 4.3.2 Implementasi fungsi main(2)

|  |  |
| --- | --- |
| 39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49. | for(it\_dfa=dfa\_graph.begin();it\_dfa != dfa\_graph.end(); it\_dfa++){  if(dfa\_lbl[it\_dfa->second].count(nfaFinish))  {  res += result[0][it\_dfa->second];  }  }  printf("%I64d\n", res%MOD\_CONSTANT);  }  return 0;  } |

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | void Initialize()  {  nid = 0;  did = 0;  memset(dfa, -1, sizeof(dfa));  memset(dfa\_lbl, 0, sizeof(dfa\_lbl));    for(int i=0;i<MAX\_NODE;i++)  {  for(int j=0;j<3;j++)  {  nfa\_nodes[i][j].clear();  }  }  while(!states.empty())  states.pop();  dfa\_graph.clear();  } |

Kode Sumber 4.3.4 Implementasi fungsi Initialize

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

Kode Sumber 4.4.1 Implementasi fungsi Preprocess

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

Kode Sumber 4.5.1 Implementasi fungsi ConvertREtoNFA (1)

|  |  |
| --- | --- |
| 7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37. | {  switch(input[i])  {  case '(':  continue;  break;  case 'a':  case 'b':  {  nfa\_nodes[nid][input[i]-'a'].push\_back(nid+1);  states.push(pair\_int(nid, nid+1));  nid+=2;  }  break;  case ')':  op = '-';  if(ops.empty() == false)  {  op = ops.top();  ops.pop();  }  switch(op)  {  case '|':  {  operan2 = states.top(); states.pop();  operan1 = states.top(); states.pop();  nfa\_nodes[nid][2].push\_back(operan1.a);  nfa\_nodes[nid][2].push\_back(operan2.a);  nfa\_nodes[operan1.b][2].push\_back(nid+1);  nfa\_nodes[operan2.b][2].push\_back(nid+1); |

Kode Sumber 4.5.2 Implementasi fungsi ConvertREtoNFA (2)

|  |  |
| --- | --- |
| 38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59.  60.  61.  62.  63.  64.  65. | states.push(pair\_int(nid, nid+1));  nid+=2;  }  break;  case '.':  {  operan2 = states.top(); states.pop();  operan1 = states.top(); states.pop();  nfa\_nodes[operan1.b][2].push\_back(operan2.a);  states.push(pair\_int(operan1.a, operan2.b));  }  break;  case '\*':  {  operan2 = states.top(); states.pop();  nfa\_nodes[nid][2].push\_back(operan2.a);  nfa\_nodes[operan2.b][2].push\_back(nid);  states.push(pair\_int(nid, nid));  nid++;  }  break;  }  break;  default:  ops.push(input[i]);  }  }  } |

Kode Sumber 4.5.3 Implementasi fungsi ConvertREtoNFA (3)

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

Kode Sumber 4.6.1 Implementasi fungsi ConvertNFAtoDFA

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | void MoveClosure(dfa\_label<int> node, int symbol, dfa\_label<int> &retval)  {  retval.clear();  for( int i=0; i<node.data; i++)  {  for(int j=0; j<nfa\_nodes[node.dt[i]][symbol].size(); j++)  { retval.push(nfa\_nodes[node.dt[i]][symbol][j]);  }  }  retval.calculateHash();  } |

Kode Sumber 4.6.2 Implementasi fungsi MoveClosure

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

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | int CreateDFAState(dfa\_label<int> label)  {  if(dfa\_graph.count(label) == 1)  return dfa\_graph[label];  for(int i=0;i<label.data;i++)  {  dfa\_lbl[did].push(label.dt[i]);  }  dfa\_lbl[did].calculateHash();  dfa\_graph[dfa\_lbl[did]] = did;  return did++;  } |

Kode Sumber 4.6.4 Implementasi fungsi CreateDFAState

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

Kode Sumber 4.7.1 Implementasi fungsi MatrixPower

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

Kode Sumber 4.7.2 Implementasi fungsi MatrixMultiply (1)

|  |  |
| --- | --- |
| 12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30. | for(i=0;i<did;i++)  {  for(j=0;j<did;j++)  {  for(k=0;k<did;k++)  {  if(mat1[i][k] == 0 || mat2[k][j] == 0 ) continue;  result[i][j]=(result[i][j]+((mat1[i][k])\*(mat2[k][j])))%MOD\_CONSTANT;  }  }  }  for(i=0;i<did;i++)  {  for(j=0;j<did;j++)  {  mat1[i][j]=result[i][j];  }  }  } |

Kode Sumber 4.7.3 Implementasi fungsi MatrixMultiply (2)

## 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 dimana 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 dimana adalah bayaknya elemen dalam *set*. Akibatnya, kompleksitas yang dibutuhkan untuk pencarian sebuah data dalam *map* dengan *set* sebagai *key* adalah .

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 . Implementasi fungsi komparasi dapat dilihat pada baris 16 s.d. 27.

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

Kode Sumber 4.8.1 Implementasi *struct* dfa\_label (1)

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

Kode Sumber 4.8.2 Implementasi *struct* dfa\_label (2)

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

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.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | struct pair\_int  {  int a;  int b;  pair\_int(){ }  pair\_int(int \_a, int \_b)  {  a = \_a;  b = \_b;  }  }; |

Kode Sumber 4.8.4 Implementasi *struct* pair\_int

***[Halaman ini sengaja dikosongkan]***

# UJI COBA DAN EVALUASI

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

## 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

* 1. Perangkat Lunak

Sistem Operasi Windows 8.0

Integrated Development Environment Code::Blocks 13.12

## 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.

### 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

### 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.

#### Pengaruh Panjang L Terhadap Waktu

Uji coba yang pertama kali dilakukan adalah dengan memberikan program beberapa panjang L yang berbeda yakni bilangan . 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 dimana *L* adalah besar pangkat dan *N* adalah dimensi matriks.

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

#### Pengaruh 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 .

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 atau bisa dibilang linear untuk setiap operator *concatenate* yang dilakukan, sedangkan untuk konstruksi NFA ke DFA diperlukan operasi sebanyak 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]***

# 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.

## 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 , 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 , untuk konversi *regular expression* menjadi NFA, dan untuk konversi NFA menjadi DFA. Kompleksitas tersebut memiliki makna jika semakin panjang *regular expression* maka semakin besar pula waktu yang dibutuhkan.

## 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

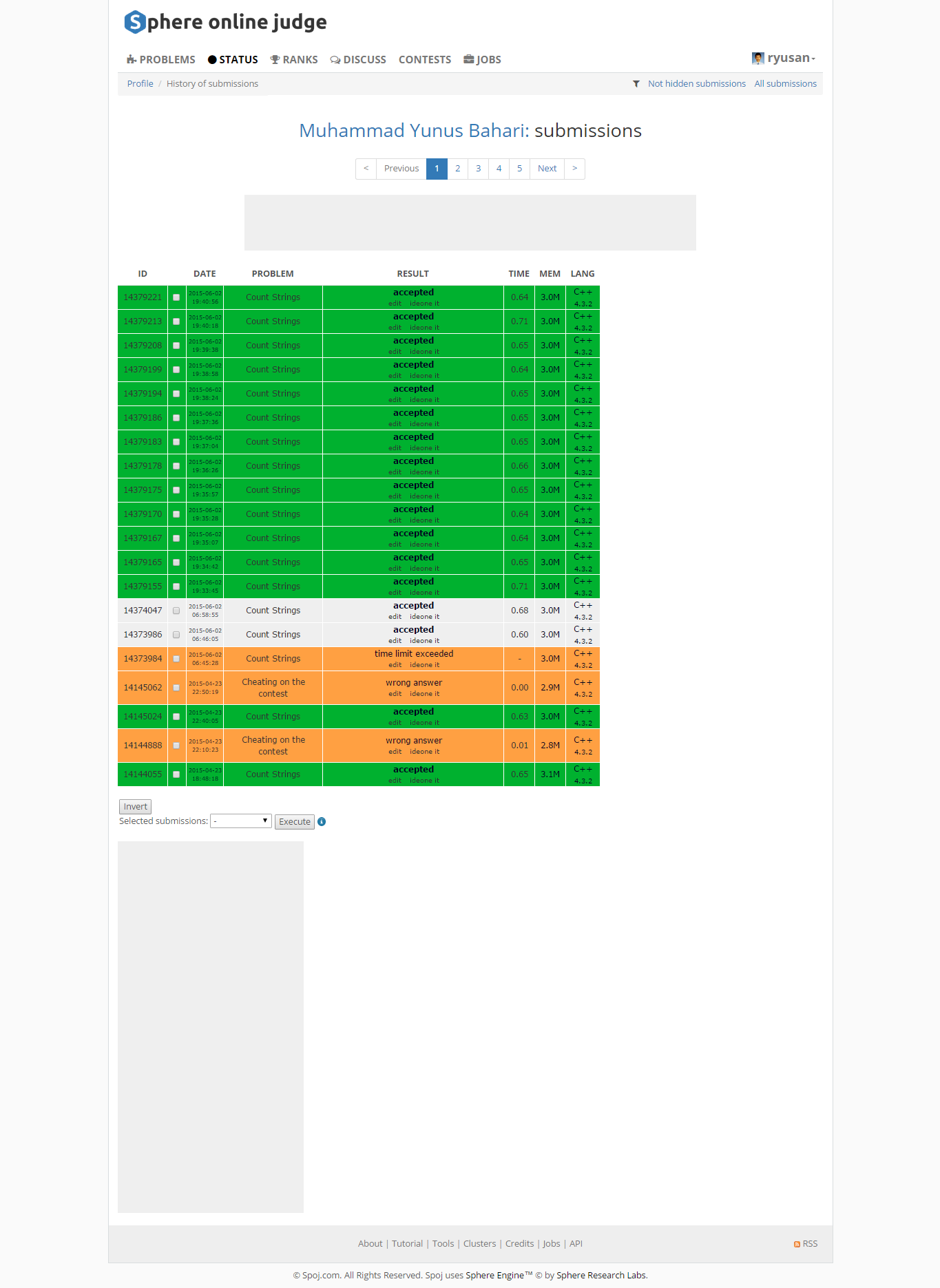
|  |  |
| --- | --- |
| [1] | G. Navaro and M. Rafinot, Flexible Pattern Matching in Strings, Cambridge: Cambridge University Press, 2002. |
| [2] | J. Power, "Constructing a DFA from an NFA (Subset Construction)," National University of Ireland, 29 November 2002. [Online]. Available: http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node9.html. [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 |



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).