

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 alfabet 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-
42. >second].count(nfaFinish))
43.         {
44.             res    +=    result[0][it_dfa-
45. >second];
46.         }
47.     }
48.     printf("%I64d\n", res%MOD_CONSTANT);
49.     return 0;
50. }

```

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]-
17. 'a'].push_back(nid+1);
18.                     states.push(pair_int(nid,
19. nid+1));
20.                     nid+=2;
21.                 }
22.                 break;
23.             case ')':
24.                 op = '-';
25.                 if(ops.empty() == false)
26.                 {
27.                     op = ops.top();
28.                     ops.pop();
29.                 }
30.                 switch(op)
31.                 {
32.                     case '|':
33.                         {
34.                             operan2 = states.top();
35.                             states.pop();
36.                             operan1 = states.top();
37.                             states.pop();
38.                             nfa_nodes[nid][2].push_back(operan1.a);
39.                             nfa_nodes[nid][2].push_back(operan2.a);
40.                             nfa_nodes[operan1.b][2].push_back(nid+1);
41.                             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();
45.         states.pop();
46.         operan1 = states.top();
47.         states.pop();
48.         nfa_nodes[operan1.b][2].push_back(operan2.a);
49.         ;
50.         states.push(pair_int(operan1.a, operan2.b));
51.         }
52.         break;
53.         case '*':
54.         {
55.             operan2 = states.top();
56.             states.pop();
57.             nfa_nodes[nid][2].push_back(operan2.a);
58.             nfa_nodes[operan2.b][2].push_back(nid);
59.             states.push(pair_int(nid, nid));
60.             nid++;
61.             }
62.             break;
63.             }
64.             break;
65.             default:
66.             ops.push(input[i]);
67.             }
68.         }
69.     }

```

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, NFAsart = state.a, tmp;
4.     stack<int> st;
5.     dfa_label<int> visited;
6.     dfa_label<int> retval;
7.     retval.push(NFAsart);
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.	{
9.	retval.push(nfa_nodes[node.dt[i]][sy
10.	mbol][j]);
11.	}
12.	}
13.	retval.calculateHash();
	}

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();
13.	i++)
14.	{
15.	tmp = nfa_nodes[t][2][i];
16.	if(retval.count(tmp) == 0)
17.	{
18.	retval.push(tmp);
19.	nodes.push(tmp);
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,
2.   long long** result)
3.   {
4.       for(int i=0;i<did;i++)
5.       {
6.           for(int j=0;j<did;j++)
7.           {
8.               result[i][j]=(i==j);
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
2.   long** mat2)
3.   {
4.       long long result[did][did];
5.       int i,j,k;
6.       for(i=0;i<did;i++)
7.       {
8.           for(j=0;j<did;j++)
9.           {
10.              result[i][j]=0LL;
11.          }
12.      }
```

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 banyaknya elemen dalam *set*. Akibatnya, kompleksitas

yang dibutuhkan untuk pencarian sebuah data dalam *map* dengan *set* sebagai *key* adalah $Nx\log_2M$.

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