

**杭州电子科技大学**

**《编译原理课程实践》**

**实验报告**

题 目：nfa转dfa

学 院：计算机

专 业：计算机科学与技术

班 级：21052314

学 号：21051408

姓 名：蔡龙祥

完成日期：2023-11-7

1. **实验目的**

学习nfa转换为dfa的方法并且通过算法得到最小化的dfa

1. **实验内容与实验要求**

NFA向DFA转化及最小化

1. **设计方案与算法描述**

题目2.基于子集法的nfa到dfa转换

1. 以nfa为基础,找到nfa的起始状态做e\_cloure操作得到起始状态
2. 以0状态作为起点,使用所有在正则表达式中出现的字符作为连接边,构造新的状态
3. 如果状态从未出现过,就列入dfa中
4. 重复步骤2直到再也没有新状态出现
5. 基于求同法和并查集算法找到等效的状态并消除构造最小dfa
6. **测试结果**

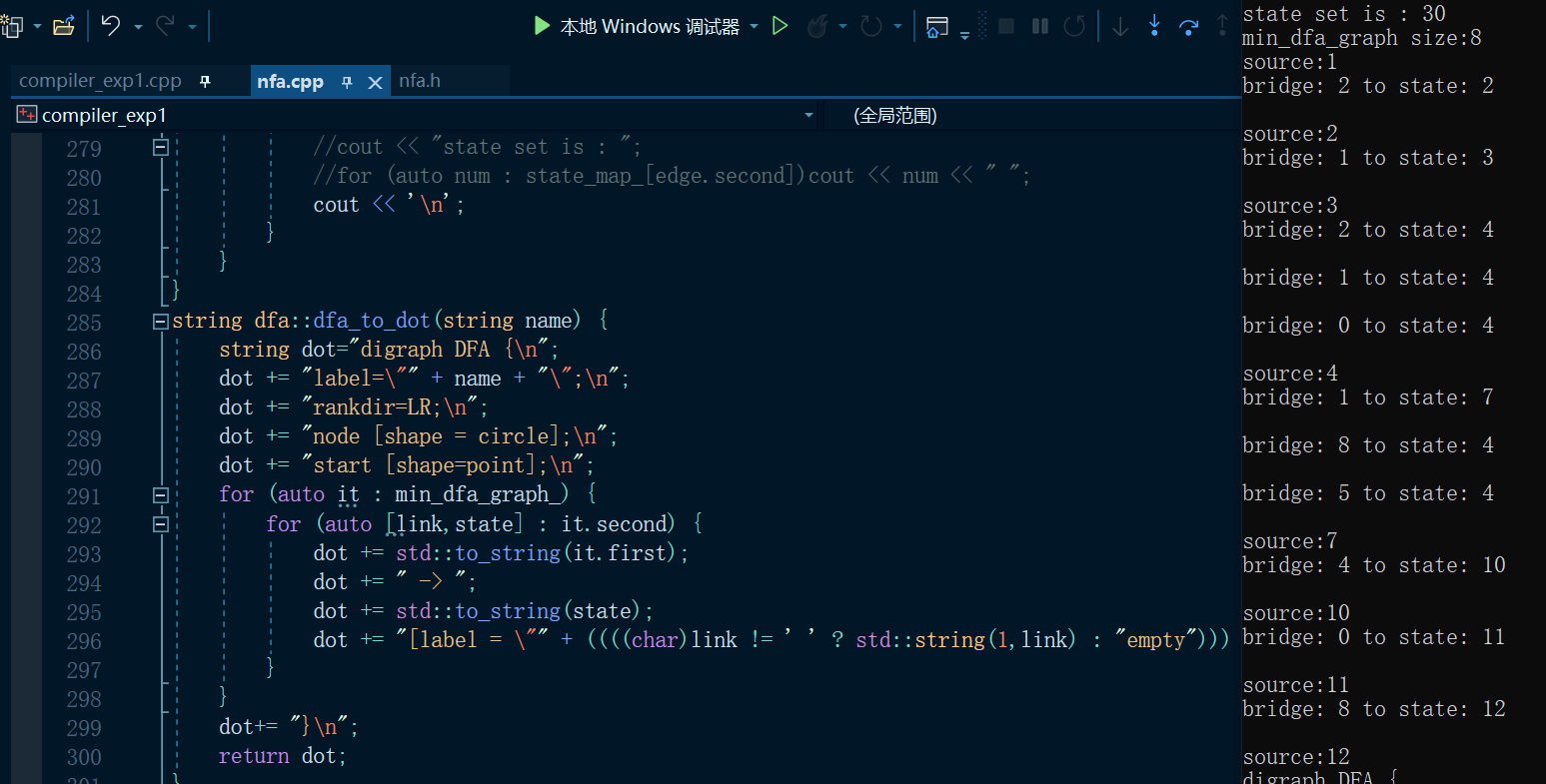


图1.构造dfa

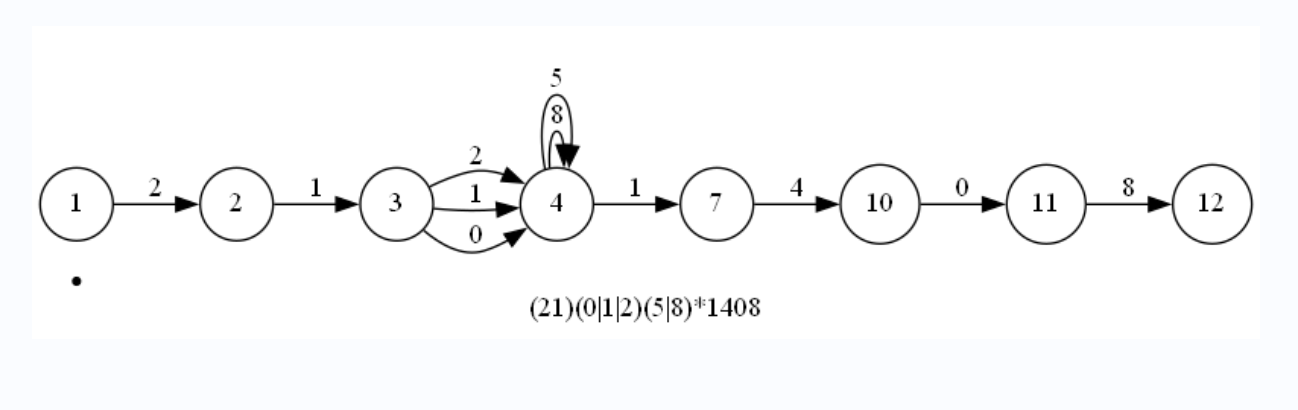


图2. dfa图示

1. **源代码**

Nfa.h

#pragma once

#include <map>

#include <utility>

#include<iostream>

#include<fstream>

#include <string.h>

#include<string>

#include <stack>

#include<iostream>

#include<vector>

#include<algorithm>

#include<queue>

#include<unordered\_set>

#include<unordered\_map>

using std::cout;

using std::string;

using std::vector;

using std::pair;

using std::make\_pair;

#define PII pair<int,int>

#define PCI pair<char,int>

#define VPI vector<PII>

#define VPC vector<PCI>

static vector<VPI>INIT\_GRAPH{ {} };

#define DEFINE\_ENUMS() \

ENUM(UNDEFINED) \

ENUM(ZERO\_ANY\_MATCH)\

ENUM(ONE\_ANY\_MATCH)\

ENUM(ZERO\_ONE\_MATCH)\

ENUM(ANY\_CHAR)\

ENUM(HEAD\_MATCH)\

ENUM(BACK\_MATCH)\

ENUM(OR)\

ENUM(LBRACE)\

ENUM(RBRACE)\

ENUM(AND)

const char AND\_CHAR = '\_';

using state = int;

template<typename T>

bool operator==(vector<T>a,vector<T>b) {

if (a.size() != b.size())return false;

int n = a.size();

for (int i = 0; i < n; i++) {

if (a[i] != b[i])return false;

}

return true;

}

struct vector\_hash {

std::size\_t operator()(std::vector<int> vec) const {

std::size\_t seed = vec.size();

for (auto& i : vec) {

seed ^= i + 0x9e3779b9 + (seed << 6) + (seed >> 2);

}

return seed;

}

};

int find(vector<int>& v, int x);

class nfa

{

public:

nfa() = default;

nfa(string name, string regex);

string name\_;

string regex\_;

string regex\_post\_;

state start\_, end\_;

vector<VPI>graph\_;

int iterator;

virtual bool build();

virtual int next();

struct char\_hash {

std::size\_t operator()(char ch) const {

size\_t seed = 0;

seed ^= ch + 0x9e3779b9 + (seed << 6) + (seed >> 2);

return seed;

}

};

std::unordered\_set<int>chs;

private:

string regex\_to\_postfix(string regex);

string regex\_post() { return regex\_post\_; }

};

enum class REGEX\_OP\_TYPE {

UNDEFINED = 'x',

ZERO\_ANY\_MATCH = '\*',

ONE\_ANY\_MATCH = '+',

ZERO\_ONE\_MATCH = '?',

ANY\_CHAR = '.',

HEAD\_MATCH = '^',

BACK\_MATCH = '$',

OR = '|',

LBRACE = '(',

RBRACE = ')',

AND = AND\_CHAR, ///< '\_'

};

enum class REGEX\_OP\_PRIORITY {

#define PRI(name,pri) name##\_PRIORITY=pri ,

PRI(UNDEFINED,-1)

PRI(ZERO\_ANY\_MATCH,3)\

PRI(ONE\_ANY\_MATCH,4)\

PRI(ZERO\_ONE\_MATCH,4)\

PRI(ANY\_CHAR,5)\

PRI(HEAD\_MATCH,6)\

PRI(BACK\_MATCH,7)\

PRI(OR,1)\

PRI(LBRACE,0)\

PRI(RBRACE,0)\

PRI(AND,2)

#undef PRI

};

REGEX\_OP\_PRIORITY regex\_op\_priority(char op);

bool priority\_lt(char l\_op, char r\_op);

class dfa :public nfa

{

public:

dfa() = default;

dfa(string name, string regex);

dfa(nfa\* nfa);

void print();

bool build();

std::unordered\_map<int, VPC>dfa\_graph\_;

std::unordered\_map<int, VPC>min\_dfa\_graph\_;

void dfs\_move(int num,char ch, vector<int>& v,vector<bool>&vis);

std::unordered\_map<int, VPC> get\_min\_dfa();

string dfa\_to\_dot(string name);

void print\_dfa(std::unordered\_map<int, VPC>&graph);

private:

nfa nfa\_;

std::unordered\_set<vector<int>,vector\_hash>state\_set\_;

std::unordered\_map<int,vector<int>>state\_map\_;

std::unordered\_map<vector<int>,int, vector\_hash>state\_map\_reverse\_;

};

Nfa.cpp

#include "nfa.h"

nfa::nfa(string name, string regex)

:name\_(name),

regex\_(regex),

start\_(0),

end\_(0),

iterator(0),

regex\_post\_(regex\_to\_postfix(regex)),

graph\_(INIT\_GRAPH)

{}

string nfa::regex\_to\_postfix(string regex) {

string tmp;

for (int i = 0; i < regex.size() - 1; i++) {

tmp += regex[i];

if (regex[i] != '(' && regex[i+1] != ')'&& regex[i] != '|' && regex[i + 1] != '|'

&& regex[i + 1] != '\*') tmp += AND\_CHAR;

}

tmp += regex.back();

std::cout << regex\_ << std::endl;

std::cout << tmp << std::endl;

std::stack<char>op\_stack;

string post\_regex;

for (auto c : tmp) {

if (isalnum(c)) {

post\_regex += c;

continue;

}

if (REGEX\_OP\_TYPE(c) == REGEX\_OP\_TYPE::LBRACE) {

op\_stack.push(c);

}

else if (REGEX\_OP\_TYPE(c) == REGEX\_OP\_TYPE::ZERO\_ANY\_MATCH

|| REGEX\_OP\_TYPE(c) == REGEX\_OP\_TYPE::AND

|| REGEX\_OP\_TYPE(c) == REGEX\_OP\_TYPE::OR){

while (op\_stack.size()) {

if (priority\_lt(op\_stack.top(), c)){

break;

}

else {

post\_regex += op\_stack.top();

op\_stack.pop();

}

}

op\_stack.push(c);

}

else if(REGEX\_OP\_TYPE(c) == REGEX\_OP\_TYPE::RBRACE) {

while (op\_stack.size() && op\_stack.top() != char(REGEX\_OP\_TYPE::LBRACE)) {

post\_regex += op\_stack.top();

op\_stack.pop();

}

op\_stack.pop();

}

}

while (op\_stack.size()) {

post\_regex += op\_stack.top();

op\_stack.pop();

}

std::cout << post\_regex << std::endl;

return post\_regex;

}

const char LINK = ' ';

int nfa::next() {

graph\_.push\_back(VPI{});

return ++iterator;

}

bool nfa::build() {

bool res = true;

std::stack<PII>edges;

int source, destination;

for (auto ch : regex\_post\_) {

if (REGEX\_OP\_TYPE(ch) != REGEX\_OP\_TYPE::ZERO\_ANY\_MATCH && REGEX\_OP\_TYPE(ch) != REGEX\_OP\_TYPE::AND&& REGEX\_OP\_TYPE(ch) != REGEX\_OP\_TYPE::OR) {

source = next();

destination = next();

edges.push(PII{ source,destination });

graph\_[source].push\_back(make\_pair(destination, ch));

chs.insert(ch);

continue;

}

switch (REGEX\_OP\_TYPE(ch)) {

case REGEX\_OP\_TYPE::ZERO\_ANY\_MATCH: {

PII edge = edges.top(); edges.pop();

source = next();

destination = next();

edges.push(PII{ source,destination });

graph\_[source].push\_back(PII{ edge.first,LINK });

graph\_[source].push\_back(PII{ destination,LINK });

graph\_[edge.second].push\_back(PII{ edge.first, LINK });

graph\_[edge.second].push\_back(PII{ destination, LINK });

}break;

case REGEX\_OP\_TYPE::AND: {

PII right = edges.top(); edges.pop();

PII left = edges.top(); edges.pop();

edges.push(PII{ left.first,right.second });

graph\_[left.second].push\_back(PII{ right.first,LINK });

}break;

case REGEX\_OP\_TYPE::OR: {

PII right = edges.top(); edges.pop();

PII left = edges.top(); edges.pop();

source = next();

destination = next();

edges.push(PII{ source,destination });

graph\_[source].push\_back(PII{left.first, LINK });

graph\_[source].push\_back(PII{right.first, LINK});

graph\_[left.second].push\_back(PII{destination,LINK});

graph\_[right.second].push\_back(PII{ destination,LINK});

}break;

default:break;

}

}

start\_ = edges.top().first;

end\_ = edges.top().second;

cout << "start from:" << start\_ << " end at:" << end\_ << "\n";

for (int i = 1; i <= iterator; i++) {

for (auto edge : graph\_[i]) {

std::cout << i << "----" << char(edge.second) << "---->" << edge.first << "\n";

}

}

return res;

}

REGEX\_OP\_PRIORITY regex\_op\_priority(char op) {

switch (REGEX\_OP\_TYPE(op)) {

#define ENUM(name) case REGEX\_OP\_TYPE::name : return REGEX\_OP\_PRIORITY::##name##\_PRIORITY;

DEFINE\_ENUMS()

#undef ENUM

default:return REGEX\_OP\_PRIORITY::UNDEFINED\_PRIORITY;

}

}

bool priority\_lt(char l\_op, char r\_op) {

return (regex\_op\_priority(l\_op) < regex\_op\_priority(r\_op));

}

dfa::dfa(string name, string regex) :nfa\_(nfa(name, regex)){}

dfa::dfa(nfa\* nfa) {

iterator=0;

nfa\_ = \*nfa;

}

void dfa::print() {

}

bool dfa::build(){

cout << "build dfa\n" ;

//子集构造法

std::queue<vector<int>>q;

vector<int>v;

v.push\_back(nfa\_.start\_);

int n = nfa\_.iterator;

vector<bool>vis(n + 1, false);

dfs\_move(nfa\_.start\_, LINK, v, vis);

v.erase(std::unique(v.begin(), v.end()),v.end());

sort(v.begin(), v.end());

cout << "start state:\n";

for (auto it : v)cout << it << " ";

cout << std::endl;

q.push(v);

state\_set\_.insert(v);

state\_map\_.insert(make\_pair(++iterator, v));

state\_map\_reverse\_.insert(make\_pair(v, iterator));

while (q.size()) {

vector<int>vec = q.front(); q.pop();

cout << "get vec:";

for (auto it : vec)cout << it << " "; cout << '\n';

for (auto ch : nfa\_.chs) {

vector<int>next;

vis = vector<bool>(n + 1, false);

for (auto num : vec) {

dfs\_move(num, ch, next, vis);

}

cout << "use char " << char(ch) << " to state:";

for (auto it : next)cout << it << " ";

cout << "\n";

vis = vector<bool>(n + 1, false);

vector<int>tmp = next;

for (auto num : tmp) {

dfs\_move(num, LINK, next, vis);

}

next.erase(std::unique(next.begin(), next.end()), next.end());

sort(next.begin(), next.end());

int source = state\_map\_reverse\_[vec];

if (next.size()&&!state\_set\_.count(next)) {

++iterator;

cout << "------------------------------------\n";

cout << "get new state:"<<iterator << std::endl;

cout << "------------------------------------\n";

state\_set\_.insert(next);

state\_map\_.insert(make\_pair(iterator, next));

state\_map\_reverse\_.insert(make\_pair(next, iterator));

q.push(next);

}

auto pos = state\_map\_reverse\_.find(next);

if (pos!= state\_map\_reverse\_.end()) {

dfa\_graph\_[source].push\_back({ ch,pos->second});

}

}

}

cout<<"dfa state\_size: "<< state\_set\_.size() << std::endl;

cout << "dfa\_graph size: " << dfa\_graph\_.size() << std::endl;

for (auto it : dfa\_graph\_) {

cout << "source:" << it.first << '\n';

for (auto edge : it.second) {

cout << "bridge: " << edge.first << " to state: " << edge.second << "\n";

cout << "state set is : ";

for (auto num : state\_map\_[edge.second])cout << num << " ";

cout << '\n';

}

}

return true;

}

void dfa::dfs\_move(int num,char ch, vector<int>& next, vector<bool>& vis) {

if (vis[num])return;

vis[num] = true;

for (auto [a, b] : nfa\_.graph\_[num]) {

//cout << "num:" << num << "\n" << "graph size:" << nfa\_.graph\_[num].size() << "\n";

if (ch == char(b)) {

next.push\_back(a);

dfs\_move(a, ch, next, vis);

}

}

}

int find(vector<int>& v, int x) {

if (x == v[x])return x;

return v[x] = find(v,v[x]);

}

std::unordered\_map<int, VPC>dfa::get\_min\_dfa() {

std::unordered\_map<int, VPC >graph=dfa\_graph\_;

int n = iterator;

vector<VPC>v(n + 1);

vector<int>f(n + 1, 0);

vector<PII>mt;

for (int i = 1; i <= n; i++)f[i] = i;

for (auto node: graph) {

v[node.first] = node.second;

}

for (int i = 1; i <= n; i++) {

for (int j = i + 1; j <= n; j++) {

if (v[i] == v[j]) {

//cout << "node[" << i << "]==node[" << j << "]\n";

mt.push\_back({ i,j });

}

}

}

for (auto [a, b] : mt) {

int fa = find(f, a);

int fb = find(f, b);

if (fa != fb) {

if (fa < fb)f[fb] = fa;

else f[fa] = f[fb];

}

}

for (int i = 1; i <= n; i++) {

if (i != f[i]) {

graph.erase(i);

continue;

}

else {

for (auto& [a, b] : graph[i]) {

if (b<=n&&b != f[b]) {

b = f[b];

}

else if(b>n)cout <<"b =="<<b<< " out of range\n";

}

}

}

cout << "min\_dfa\_graph size:" << graph.size() << std::endl;

print\_dfa(graph);

min\_dfa\_graph\_ = graph;

return graph;

}

void dfa::print\_dfa(std::unordered\_map<int, VPC>&graph) {

for (auto it : graph) {

cout << "source:" << it.first << '\n';

for (auto [link, state] : it.second) {

cout << "bridge: " << ((((char)link != ' ' ? std::string(1, link) : "empty"))) << " to state: " << state << "\n";

//cout << "state set is : ";

//for (auto num : state\_map\_[edge.second])cout << num << " ";

cout << '\n';

}

}

}

string dfa::dfa\_to\_dot(string name) {

string dot="digraph DFA {\n";

dot += "label=\"" + name + "\";\n";

dot += "rankdir=LR;\n";

dot += "node [shape = circle];\n";

dot += "start [shape=point];\n";

for (auto it : min\_dfa\_graph\_) {

for (auto [link,state] : it.second) {

dot += std::to\_string(it.first);

dot += " -> ";

dot += std::to\_string(state);

dot += "[label = \"" + ((((char)link != ' ' ? std::string(1,link) : "empty"))) + "\"]; \n";

}

}

dot+= "}\n";

return dot;

}