一、实验目的

掌握 NFA 到 DFA 转换的方法和过程

二、 实验内容与实验要求

二、实验内容

1.输入一个 NFA，输出一个接受同一正规集的 DFA；

2 编制测试程序；

3.调试程序

三、实验要求

1.采用任意语言，实现该算法；

2.上交源代码；

3.撰写实验报告，在实验报告里要写清楚测试结果，并以 word 形式上交实验报告；

三、 设计方案与算法描述

采用子集法构造 dfa

伪代码如下

```
Subset-Construction(NFA)
  let Dtran be a table
  DFA_States = {ε-closure(NFA.s0)}  # DFA 状态集合的初始状态为 NFA 初始状态的
闭包，并且未标记
  while (exist T in DFA_States not marked) { # 存在未标记的 DFA 状态
    mark T  # 标记 T，表示查过 T 状态的所有后续状态了
    for (a in Σ) {
      Tc = ε-closure(move(T, a)) # 找到所有输入字符对应的下一个状态
```

```
            if (Tc not in DFA_States) { # 将状态加入到 DFA_States
                push Tc in DFA_States & unmarked Tc
            }
            Dtran[T, a] = Tc
        }
    }
    return Dtran
```

四、 测试结果

测试例子

```
#a|b
start: 0
end: 3
count: 6
0 1 #
1 2 a
2 3 #
0 4 #
4 5 b
5 3 #

#aba
start: 0
end: 5
count: 6
0 1 a
1 2 #
2 3 b
3 4 #
4 5 a

# (a|b)*
start: 0
end: 5
count: 8
0 1 #
1 2 #
2 3 a
3 4 #
4 1 #
4 5 #
1 6 #
```

6 7 b
7 4 #
0 5 #

bb(a|b)*a
start: 0
end: 11
count: 14
0 1 b
1 2 #
2 3 b
3 4 #
4 5 #
5 6 #
6 7 a
7 8 #
8 5 #
8 9 #
9 10 #
10 11 a
5 12 #
12 13 b
13 8 #
4 9 #

结果

```
❯ ./target/main test/*
start: 0
end: 2,1
count: 3
0--b-->1
0--a-->2


start: 0
end: 3
count: 4
0--a-->0
1--b-->2
2--a-->3


start: 0
end: 0,2,1
count: 3
0--b-->1
0--a-->2
1--b-->1
1--a-->2
2--b-->1
2--a-->2


start: 0
end: 3
count: 5
0--b-->1
1--b-->2
2--a-->3
2--b-->4
3--a-->3
3--b-->4
4--a-->3
4--b-->4
```

五、 源代码

```
#include "../../common/comm.hpp"
#include <algorithm>
#include <cassert>
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <functional>
#include <initializer_list>
#include <iostream>
#include <iterator>
#include <optional>
#include <ostream>
#include <regex>
#include <set>
#include <stack>
#include <string>
#include <string_view>
#include <tuple>
#include <unordered_map>
#include <utility>
#include <vector>

using std::function;
using std::make_pair;
using std::optional;
using std::set;
using std::sregex_token_iterator;
using std::stack;
using std::stoi;
using std::string_view;
using std::unordered_map;
using std::vector;

struct Bitset {
  uint64_t content;
  Bitset() : content(0) {}
  Bitset(uint64_t content) : content(content) {}
  Bitset(const Bitset &bitset) : content(bitset.content) {}
  Bitset &operator=(const Bitset &bitset) {
    content = bitset.content;
```

```
  return *this;
}
Bitset(std::initializer_list<int> list) {
  *this = 0;
  for (auto item : list) {
    this->insert(item);
  }
}

Bitset operator|(const Bitset &bitset) const {
  return Bitset(content | bitset.content);
}
Bitset operator&(const Bitset &bitset) const {
  return Bitset(content & bitset.content);
}
Bitset operator~() const { return Bitset(~content); }
Bitset operator^(const Bitset &bitset) const {
  return Bitset(content ^ bitset.content);
}
Bitset &operator|=(const Bitset &bitset) {
  content |= bitset.content;
  return *this;
}
Bitset &operator&=(const Bitset &bitset) {
  content &= bitset.content;
  return *this;
}
Bitset &operator^=(const Bitset &bitset) {
  content ^= bitset.content;
  return *this;
}
bool operator==(const Bitset &bitset) const {
  return content == bitset.content;
}
bool operator!=(const Bitset &bitset) const {
  return content != bitset.content;
}
bool operator<(const Bitset &bitset) const {
  return content < bitset.content;
}
bool operator>(const Bitset &bitset) const {
  return content > bitset.content;
}
bool operator<=(const Bitset &bitset) const {
```

```
  return content <= bitset.content;
}
bool operator>=(const Bitset &bitset) const {
  return content >= bitset.content;
}

bool contains(int i) const {
  assert(i >= 0 && i < 64);
  return (content & (1ULL << i)) != 0;
}

bool empty() const { return content == 0; }

Bitset insert(int i) {
  assert(i >= 0 && i < 64);
  content |= (1ULL << i);
  return *this;
}

struct Iterator {
  uint64_t content;
  int index;
  Iterator(uint64_t content, int index) : content(content), index(index) {}
  Iterator &operator++() {
    ++index;
    content >>= 1;
    while (content != 0 && (content & 1) == 0) {
      ++index;
      content >>= 1;
    }
    return *this;
  }
  Iterator operator++(int) {
    Iterator tmp = *this;
    ++*this;
    return tmp;
  }
  bool operator==(const Iterator &it) const { return content == it.content; }
  bool operator!=(const Iterator &it) const { return !(*this == it); }
  int operator*() const { return index; }
};

Iterator begin() const {
  uint64_t tmp = content;
```

```
    int index = 0;
    while (tmp != 0 && (tmp & 1) == 0) {
      ++index;
      tmp >>= 1;
    }
    return Iterator(tmp, index);
  }

  Iterator end() const { return Iterator(0, 64); }

  static Bitset from_string(string_view str) {
    auto bitset = Bitset();
    assert(str.front() == '{' && str.back() == '}');
    str.remove_prefix(1);
    str.remove_suffix(1);
    while (!str.empty()) {
      auto pos = str.find(',');
      if (pos == string_view::npos) {
        pos = str.size();
      }
      auto num = stoi(std::string(str.substr(0, pos)));
      bitset.insert(num);
      str = str.substr(pos < str.size() ? pos + 1 : pos);
    }
    return bitset;
  }

  std::string to_string() {
    auto ret = std::string();
    for (auto i : *this) {
      ret += std::to_string(i) + ",";
    }
    if (!ret.empty()) {
      ret.pop_back();
    }
    return "{" + ret + "}";
  }
};

constexpr uint64_t MAX_NFA_STATE = sizeof(uint64_t) * 8;
constexpr char EPSILON = '#';

struct NFA {
  struct State {
```

```
  // 对于一个节点和给定的输入符号, 可以转移到的下一个节点的集合的映射
  using Trans = unordered_map<char, Bitset>;
  Trans to;
};

// 起始状态的编号

int start;

// 终止状态的编号

int end;

// 状态集合

vector<State> states;
std::string symbols;
NFA(int start, int end, int total) : start(start), end(end), states(total) {}


Bitset epsilon_closure(int id) {
  auto closure = Bitset{id};
  auto s = stack<int>();
  s.push(id);
  while (!s.empty()) {
    auto state = s.top();
    s.pop();
    for (auto i : this->states.at(state).to[EPSILON]) {
      if (!closure.contains(i)) {
        closure.insert(i);
        s.push(i);
      }
    }
  }
  return closure;
}
Bitset epsilon_closure(Bitset set) {
  auto closure = set;
  auto s = stack<int>();
  for (auto i : set) {
    s.push(i);
  }
  while (!s.empty()) {
    auto state = s.top();
    s.pop();
    for (auto i : this->states.at(state).to[EPSILON]) {
      if (!closure.contains(i)) {
```

```
        closure.insert(i);
        s.push(i);
      }
    }
  }
  return closure;
}

Bitset move(int id, char c) { return this->states.at(id).to[c]; }
Bitset move(Bitset set, char c) {
  auto move_set = Bitset();
  for (auto i : set) {
    move_set |= this->states.at(i).to[c];
  }
  return move_set;
}

//<start>
//<end>
//<total>
//<symbols> ...
//<trans> ...
//...

static NFA *from_str(string_view str) {
  auto lines = Util::lines(str);
  assert(lines.size() > 3);
  auto start_line = lines[0];
  auto end_line = lines[1];
  auto count_line = lines[2];
  constexpr string_view START_PREFIX = "start: ";
  constexpr string_view END_PREFIX = "end: ";
  constexpr string_view COUNT_PREFIX = "count: ";
  assert(start_line.substr(0, START_PREFIX.length()) == START_PREFIX);
  assert(end_line.substr(0, END_PREFIX.length()) == END_PREFIX);
  assert(count_line.substr(0, COUNT_PREFIX.length()) == COUNT_PREFIX);
  start_line = start_line.substr(START_PREFIX.length());
  end_line = end_line.substr(END_PREFIX.length());
  count_line = count_line.substr(COUNT_PREFIX.length());

  auto start = Util::string_view2int(start_line);
  auto end = Util::string_view2int(end_line);
  auto count = Util::string_view2int(count_line);
  auto nfa = new NFA(start, end, count);
```

```
    auto symbols = std::string();

    using Tran = std::tuple<int, char, int>;

    auto extract = [](string_view line) -> Tran {
      auto tokens = Util::split(line, ' ');
      assert(tokens.size() == 3);
      assert(tokens[2].length() == 1);
      auto from = Util::string_view2int(tokens[0]);
      auto to = Util::string_view2int(tokens[1]);
      auto symbol = tokens[2].front();
      return {from, symbol, to};
    };

    for (int i = 3; i < lines.size(); i++) {
      auto [from, symbol, to] = extract(lines[i]);
      nfa->states.at(from).to[symbol].insert(to);
      if (symbol != EPSILON && symbols.find(symbol) == std::string::npos) {
        symbols += symbol;
      }
    }
    nfa->symbols = symbols;
    return nfa;
  }
};

struct DFA {
  struct State {
    static constexpr int UNALLOCID = -1;
    int id;

    // 对于 dfa state 对应 nfa state 的一个子集

    Bitset nfa_states;
    typedef unordered_map<char, State *> Trans;
    Trans to;
    State() : nfa_states(), to(), id(UNALLOCID) {}
    State(Bitset nfa_states) : nfa_states(nfa_states), to(), id(UNALLOCID) {}

    bool operator==(const State &state) const {
      return nfa_states == state.nfa_states;
    }
    bool operator!=(const State &state) const { return !(*this == state); }
    bool operator<(const State &state) const {
      return nfa_states < state.nfa_states;
```

```
  }
  bool operator>(const State &state) const {
    return nfa_states > state.nfa_states;
  }
  bool operator<=(const State &state) const {
    return nfa_states <= state.nfa_states;
  }
  bool operator>=(const State &state) const {
    return nfa_states >= state.nfa_states;
  }

  void visit(function<void(State &)> func, set<State *> &has_visited) {
    func(*this);
    has_visited.insert(this);
    for (auto [_, s] : this->to) {
      if (has_visited.find(s) == has_visited.end()) {
        s->visit(func, has_visited);
      }
    }
  }
};
State *start;
set<State *> ends;
set<State *> states;
std::string symbols;

DFA(State *s0, std::string symbols)
    : start(s0), ends(), states(), symbols(symbols) {
  states.insert(s0);
  auto pos = this->symbols.find('#');
  if (pos != std::string::npos) {
    this->symbols.erase(pos);
  }
}

optional<State *> find_state(function<bool(const State &s)> pred) {
  for (auto state : states) {
    if (pred(*state)) {
      return state;
    }
  }
  return {};
}
```

```
// 确定终态集
void set_end_states(int dfa_end) {
  for (auto state : this->states) {
    if (state->nfa_states.contains(dfa_end)) {
      this->ends.insert(state);
    }
  }
}

static DFA *from_nfa(NFA &nfa) {
  auto s0 = new State(nfa.epsilon_closure(0));
  auto dfa = new DFA(s0, nfa.symbols);
  auto to_solve = stack<State *>();
  to_solve.push(s0);

  while (!to_solve.empty()) {
    auto state = to_solve.top();
    to_solve.pop();
    for (auto symbol : dfa->symbols) {
      auto move_closure =
          nfa.epsilon_closure(nfa.move(state->nfa_states, symbol));
      if (!move_closure.empty()) {
        if (auto to = dfa->find_state([=](const State &state) -> bool {
              return state.nfa_states == move_closure;
            });
            to.has_value()) {
          state->to.insert(make_pair(symbol, to.value()));
        } else {
          auto new_state = new State(move_closure);
          state->to.insert(make_pair(symbol, new_state));
          to_solve.push(new_state);
          dfa->states.insert(new_state);
        }
      }
    }
  }
  dfa->set_end_states(nfa.end);
  auto allocator = 0;
  auto has_visited = set<State *>();
  dfa->start->visit([&](State &s) { s.id = allocator++; }, has_visited);
  return dfa;
}
```

```
  std::string to_string() {
    auto ret = "start: " + std::to_string(this->start->id) + "\n";
    ret += "end: ";
    auto ends = std::string();
    for (auto end : this->ends) {
      ends += std::to_string(end->id) + ",";
    }
    if (!ends.empty()) {
      ends.pop_back();
    }
    ret += ends + "\n";
    ret += "count: " + std::to_string(this->states.size()) + "\n";
    auto has_visited = set<State *>();
    this->start->visit(
        [&](State &s) {
          for (auto [ch, to] : s.to) {
            ret += std::to_string(s.id) + "--" + ch + "-->" +
                   std::to_string(to->id) + "\n";
          }
        },
        has_visited);
    return ret;
  }
};
#include "./nfa_to_dfa.hpp"
#include <cassert>
#include <cstdio>
#include <iostream>

int main(int argc, char *argv[]) {
  assert(argc > 1);
  for (int i = 1; i < argc; i++) {
    auto line = argv[i];
    auto nfa = NFA::from_str(Util::read_file_to_string(line));
    std::cout << DFA::from_nfa(*nfa)->to_string() << std::endl;
    getchar();
  }
}
```