课程实验: NFA 到 DFA 的转化

0. 小组成员

- A 算法设计及撰写报告
- B 参与程序代码的实现
- C 参与程序代码的实现
- D 参与程序代码的实现
- Github 项目地址

1. 实验目的

- 编程实现 NFA 到 DFA 的转化，理解不同自动机的转化过程。

2. 实验内容

- 有限状态自动机是描述控制过程有力工具。有限状态自动机有不同的类型，例如，确定有限状态自动机（DFA）和不确定有限状态自动机（NFA）。这些不同类型的自动机之间可以等价转化。我们在实际应用中，可以利用某种类型的自动机更加方便刻画实际系统，然后再利用等价转化算法实现不同类型的自动机转化。
- 本实验要求编程实现NFA到DFA的自动转化。输入自己设定的不确定有限自动机描述格式，输出对应的确定有限自动机。

3. 实验环境描述

- 操作系统 Windows 11 专业版 64 位, Kernel: 10.0.22000.0
- C++ 编译器 GCC version 11.2.0 (MinGW-W64 x86_64-ucrt-posix-seh, built by Brecht Sanders)
- 代码编辑器 VIM - Vi IMproved 8.2 (2019 Dec 12, compiled Sep 21 2021 16:13:20)

4. 设计思路

- 我们采用一个有向图来存储初始的 NFA, 在代码实现上，使用 C++ STL 中的各种数据结构实现。
- 核心算法：子集构造法，对于给定初始输入的 NFA, 我们先枚举其状态的所有子集，这需要指数级别的时间复杂度和空间复杂度算法，假设我们初始 NFA 的状态数为 n, 那么包括空集合在内的子集数量为 2^n 级别。
- 枚举出所有子集之后，对于每个子集我们分别计算出其所有的可达状态，然后存储下来。
- 上述步骤之后，子集构造法就基本完成了，我们还可以进一步对构造出的 DFA 进行简化。

5.核心算法

我们直接通过代码细节来逐步讲解，就以上课的课件中的此例图示：

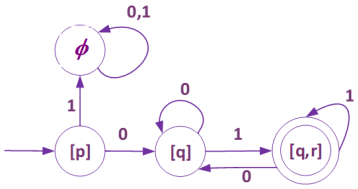
1、初始的NFA

	0	1
→ p	{q}	ϕ
q	{q}	{q, r}
* r	ϕ	ϕ

2、子集构造，计算状态可达

	0	1
ϕ	ϕ	ϕ
→ [p]	[q]	ϕ
[q]	[q]	[q, r]
* [r]	ϕ	ϕ
[p, q]	[q]	[q, r]
*[p, r]	[q]	ϕ
*[q, r]	[q]	[q, r]
*[p, q, r]	[q]	[q, r]

3、经筛选后的DFA



5.1 初始建立 NFA 所需数据结构的定义

```
set<string> T; //字符集
vector<string> S; //NFA初始状态集合

int T_SIZE; //字符集大小
int STATE_SIZE; //NFA状态数量

cin >> T_SIZE >> STATE_SIZE;
map<pair<string, string>, vector<string>> mp; //哈希表存储所有初始的集合可达状态
```

上述最后一行使用的 `std::map` 到底存什么呢？举个例子，对于图示左上角初始的 NFA, 对于状态 q, 我们输入 1，那么其转移可达的状态集合为 $\{q, r\}$，这样我们很容易发现，每个状态与其每一个输入，其可达状态都是一个唯一的集合，这样我们可以使用哈希表来存储整个逻辑结构，哈希表的 KEY 值存储状态和其输入的二元组，我们在上述代码中使用 `std::pair` 来实现，哈希表的 VALUE 值就是其可达的状态集合，比如上述的集合 $\{q, r\}$。

5.2 数据的读入

```
for (int i = 0; i < STATE_SIZE; i++) {
    string state; //具体状态
    cin >> state;
    S.push_back(state);
    for (int i = 0; i < T_SIZE; i++) { //当前状态对于所有字符集能到的状态
        string alpha;
        cin >> alpha;
        T.insert(alpha);
        int set_size;
        cin >> set_size;
        for (int i = 0; i < set_size; i++) {
            string single_state;
            cin >> single_state;
            mp[make_pair(state, alpha)].push_back(single_state);
        }
    }
}
```

这个读入的逻辑看起来有点复杂，其实不然，我们继续对着图示的例子来看，我们先输入 NFA 字符集的大小以及 NFA 状态的数量，上述图示字符集为 $\{0, 1\}$ 集合大小为 2, 初始的状态集合为 $\{p, q, r\}$ 集合大小为 3。接着开始读入每个状态以及对于该状态，每个字符集中的元素输入后其可达的状态集合，输入之后我们全部存入哈希表中。对于详细的输入输出样例，我们会在后续使用样例详细说明。

5.3 预处理枚举出所有子集

```cpp
//枚举所有子集
vector<vector<string>> ALL(1 << STATE_SIZE, vector<string>());
for (int i = 0; i < (1 << STATE_SIZE); i++) {
    for (int k = 0; k < STATE_SIZE;  k++) {
        if ((i >> k) & 1) {
            ALL[i].push_back(S[k]);
        }
    }
}

//排序便于输出
sort(ALL.begin(), ALL.end(), [&](vector<string> T1, vector<string> T2) {
    return T1.size() < T2.size();
});
```

这里的枚举代码看起来有点 tricky，其实就是使用了位运算来实现，这似乎是个很常见的技巧。我们把所有枚举出的子集都存入一个命名为 `ALL` 的可变二维数组中，而且为了后续输出的直观，我们按照字典序把状态排序，这里的排序语句使用了 C++ 中的 lambda 表达式，所以编译器需支持 C++11 及其以后的标准用于编译。

5.4 计算出所有枚举后子集的可达状态并输出

```cpp
for (auto t : ALL) { //遍历所有子集 [p,q,r] [q,r]...
    for (auto set_ele : t) cout << set_ele << ' ';
    cout << "#: ";
    for (auto c : T) { //0, 1 输入
        set<string> temp_union; //[p,0] [q,0] [r,0] union
        for (auto j : t) { //枚举所有字符集  [p,0] [q,0] [r,0]  / [p,1][q,1] [r,1]
            auto v = mp[make_pair(j, c)];
            for (auto ele : v) temp_union.insert(ele);
        }
        if (temp_union.size() > 1 && temp_union.count("NULL"))
            temp_union.erase("NULL");
        for (auto union_ele : temp_union) cout << union_ele << ' ';
        cout << "| ";
    }
    cout << endl;
}
```

这里就是最后的计算且输出阶段了，我们对于每个子集，都分别输入字符集中所有的元素来得到这个子集的所有输出，以图示为例，假设对于子集 $\{p,q,r\}$, 输入 1 之后就转移到了状态 $\{q,r\}$, 输入 0 之后就转移到了状态 q, 我们使用字符串 "NULL" 来表示空集，最后我们还要去重，因为一个子集中的转移后续状态的某个元素，能由多个该子集中的元素转移得到。上述代码的注释很清楚的描述了对状态的处理细节。

6. 程序的输入输出和执行效果

在终端编译程序的执行效果：

```
❯ g++ main.cpp -std=c++17 -static -o test && ./test.exe

2 3
p 0 1 q 1 1 NULL
```

```
q 0 1 q 1 2  q r
r 0 1 NULL 1 1 NULL

#: | |
p #: q | NULL |
q #: q | q r |
r #: NULL | NULL |
p q #: q | q r |
p r #: q | NULL |
q r #: q | q r |
p q r #: q | q r |
```

上述执行效果的输入和输出样例分开如下：

Example Input:

```
2 3
p 0 1 q 1 1 NULL
q 0 1 q 1 2  q r
r 0 1 NULL 1 1 NULL
```

Example Output:

```
#:     |         |
p #: q | NULL    |
q #: q | q r     |
r #: NULL | NULL |
p q #: q | q r   |
p r #: q | NULL  |
q r #: q | q r   |
p q r #: q | q r |
```

这里样例的输入输出与课件 PPT 所对应的 NFA 转化后得到的 DFA 的状态表示完全一致，第一行没有输出状态代表了全为空集的第一行结果。

输入样例的详细解释：对照图示来看，对于初始 DFA, 状态 p 遇到 0 会转移到集合大小为 1 的 q 状态，遇到输入 1 会转移到空集，这里我们依然用 1 大小来代表空元素，这样便于代码的实现。对于状态 q, r 也是同理。至此整个子集构造法的过程就结束了。

7. 改进思路和方法

对于 NFA 到 DFA 的转换算是一个很经典的问题，实现的方法也非常多，我 Google 了不少学术相关的文档理解了他们的不同实现思路[1]，有些思路很复杂有些很简洁，比如枚举子集构造法就非常简洁，但是这种方法并算不上最高效，因为这个算法的时间复杂度下界是严格的 $O(2^n)$。还有一些人尝试使用了深度优先搜索（Depth-first search）以及非常多的搜索剪枝技巧，判掉了很多的 corner case 来对算法进行常数复杂度的优化，这样确实对于一些特殊的输入样例能有明显效果，但是对于随机的数据，其均摊复杂度可能不会比子集构造法快多少，毕竟这个问题属于 PSPACE, 而且是一个 PSPACE-Complete 问题[2], 而且 DFA 最小化问题只属于 NL。还有很多 TCS 学术界所已有的优化方法，比如 Brzozowski's Algorithm[3] 等等, 见最后的参考文献中，这些细枝末节的优化技巧对于一个 PSPACE-Complete 问题来说降低其各种算法的 bound 确实微不足道，当然，这不排除有可能有一天会有人能够证明 P = PSPACE。

Reference:

[1] Write a program to convert NFA to DFA

[2] Complexity of NFA to DFA minimization with binary threshold

[3] Algorithm for converting very large NFA to DFA

[4] Can we efficiently convert from NFA to smallest equivalent DFA?

[5] Introduction To The Theory Of Computation - Michael Sipser

最后附上完整的子集构造法实现代码如下:

```cpp
#include <bits/stdc++.h>
using namespace std;
// #define DEBUG2

void solve() {
    set<string> T; //字符集
    vector<string> S; //NFA初始状态集合

    int T_SIZE; //字符集大小
    int STATE_SIZE; //NFA状态数量

    cin >> T_SIZE >> STATE_SIZE;
    map<pair<string, string>, vector<string>> mp;
    for (int i = 0; i < STATE_SIZE; i++) {
        string state; //具体状态
        cin >> state;
        S.push_back(state);
        for (int i = 0; i < T_SIZE; i++) { //当前状态对于所有字符集能到的状态
            string alpha;
            cin >> alpha;
            T.insert(alpha);
            int set_size;
            cin >> set_size;
            for (int i = 0; i < set_size; i++) {
                string single_state;
                cin >> single_state;
                mp[make_pair(state, alpha)].push_back(single_state);
            }
        }
    }

#ifdef DEBUG1
    for (auto t : mp) {
        auto pp = t.first;
        auto collection = t.second;
        cout << pp.first << ' ' <<  pp.second << endl;
        for (auto x : collection) cout << x << ' ';
        cout << endl;
    }
#endif

    //枚举所有子集
    vector<vector<string>> ALL(1 << STATE_SIZE, vector<string>());
```

```
    for (int i = 0; i < (1 << STATE_SIZE); i++) {
        for (int k = 0; k < STATE_SIZE;  k++) {
            if ((i >> k) & 1) {
                ALL[i].push_back(S[k]);
            }
        }
    }

    //排序便于输出
    sort(ALL.begin(), ALL.end(), [&](vector<string> T1, vector<string> T2) {
        return T1.size() < T2.size();
    });

#ifdef DEBUG2
    sort(ALL.begin(), ALL.end(), [&](vector<string> T1, vector<string> T2) {
        return T1.size() < T2.size();
    });
    for (auto t : ALL) {
        for (auto j : t) {
            cout << j << ' ';
        }
        cout << endl;
    }
#endif

    vector<vector<string>> reachable_state; //所有可达状态

    for (auto t : ALL) { //遍历所有子集 [p,q,r] [q,r]...
        for (auto set_ele : t) cout << set_ele << ' ';
        cout << "#: ";
        for (auto c : T) { //0, 1 输入
            set<string> temp_union; //[p,0] [q,0] [r,0] union
            for (auto j : t) { //枚举所有字符集  [p,0] [q,0] [r,0]  / [p,1] [q,1][r,1]
                auto v = mp[make_pair(j, c)];
                for (auto ele : v) temp_union.insert(ele);
            }
            if (temp_union.size() > 1 && temp_union.count("NULL"))
                temp_union.erase("NULL");
            for (auto union_ele : temp_union) cout << union_ele << ' ';
            cout << "| ";
        }
        cout << endl;
    }

}

int main() {
    int t = 1;
    // cin >> t;
    while (t --) solve();
    return 0;
}
```