NFA转换DFA实验

一、小组成员及分工

1. 李祥宇 2020211375 负责进行子集构造法的编写和各个模块的整合调试。
2. 李润杰 2020211374 负责进行所有集合的初始化。
3. 马天成 2020211376 负责进行状态集合的重命名。
4. 孟宇航 2020211377 负责进行最终集合的输出转换和去重。

二、实验环境

使用语言：C++

编译环境：Visual Studio 2019

三、主要的数据结构

```
class part {
public:
    string source;
    char edge ;
    string distance;
};
```

part类是负责存储输入的NFA的生成式，本程序的NFA是通过输入生成式来表示的。

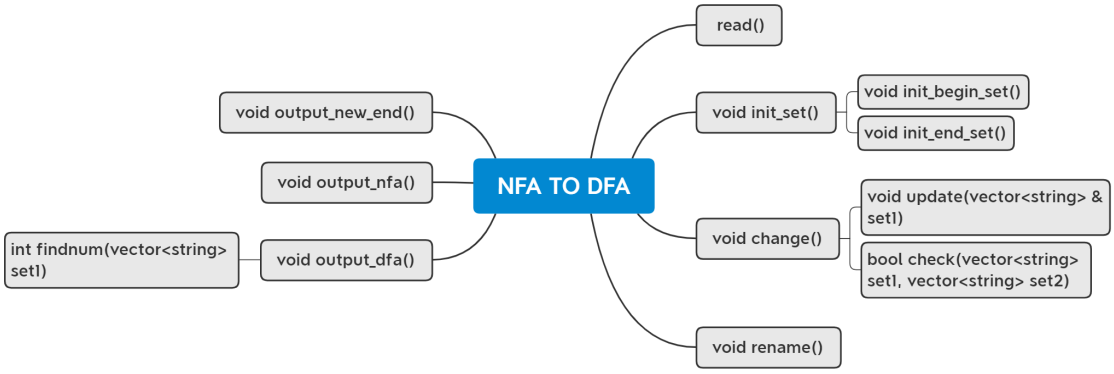
```
class NFA {
public:
    vector<char> alphabet;//字母表
    vector<part> allpart;//转换函数
    vector<string> set;//状态集合
    vector<string> begin_set;//初始集合
    vector<string> end_set;//终止集合
    vector<string> new_set;//新产生的集合
    vector<vector<string>> allset;//所有集合
    vector<int> new_name;//新的状态子集重命名
    int part_num;
    int alphabet_num;
void init_set();//初始化状态集合
void init_begin_set();//初始化初始集合
void init_end_set();//初始化终止集合
void change();//子集构造
bool check(vector<string> set1, vector<string> set2);//判断集合是否相同
void update(vector<string> &set1);//去掉状态子集中重复的状态
void rename();//重命名
void output_dfa();//输出最终形式
void output_nfa();
int findnum(vector<string> set1);//寻找在新命名的状态中的位置
void read();
void output_new_end();//输出新的终止状态集合
};
```

NFA类是将NFA转换成DFA的相关操作，每一个集合都是通过vector进行存储的，在该类中还包括了初始化相关的状态集合，子集构造法的实现以及状态子集的重命名，去重等相关操作。当然本程序NFA的输入是通过文件进行输入的，用户需要直接在文件中输入NFA的生成式以及字母表中的字母即可，在启动程序后，用户需要将原NFA的初始状态集合和终止状态集合进行输入。

四、程序的设计思路及核心算法

4.1 程序的设计思路

该程序的实现使用的方法是子集构造法，从最开始的初始状态集合开始进行运算，每次都将新产生的状态子集保存下来，并且作为新的迭代对象进行操作，通过这样的不断迭代，一直到最终没有新的状态子集产生。这样在产生了新的完整的状态转移表后，对每一个新产生的状态从0开始进行重命名，并且将终止状态进行更新。最后将新命名的DFA进行输出，同时将新的终止状态进行输出，相关函数的调用关系如下图所示。



4.2 核心算法

4.2.1 void change() 子集构造法的算法实现

```
void NFA::change()
{
    int i = 0, j;
    int lenth;
    bool flag = true;
    bool flag1 = false;
    allset.push_back(begin_set);
    while (flag)
    {
        if (allset[i].size() != 0 )
        {
            if (allset[i][allset[i].size() - 1].compare("repeat") != 0)
            {
                for (int n = 0; n <= alphabet.size() - 1; n++)
                {
                    for (j = 0; j <= allset[i].size() - 1; j++)//遍历当前集合中的状
态
                    {
                        for (int m = 0; m <= allpart.size() - 1; m++)//遍历整个转
移函数进行运算
                        {
                            if (allpart[m].source.compare(allset[i][j]) == 0 &&
allpart[m].edge == alphabet[n])
```

```
                            {
                                new_set.push_back(allpart[m].distance);
                            }
                        }
                    }
                    update(new_set);
                    for (int q = 0; q <= allset.size() - 1; q++)//检查新产生的状态子集是否与之前的有重复
                    {
                        if (check(allset[q], new_set))
                        {
                            flag1 = true;
                            new_set.push_back("repeat");//添加一个标志位，以表示重复
                            break;
                        }
                    }
                    allset.push_back(new_set);
                    lenth = new_set.size();
                    for (int p = 0; p <= lenth - 1; p++)//清空中间集合，方便新的内容进行添加
                    {
                        new_set.pop_back();
                    }
                }
            }
        }
        i++;
        if (i==allset.size())
        {
            flag = false;
        }
    }
}
```

子集构造法的核心思路就是每次把新产生的状态进行记录并迭代，整体的流程是，首先将原NFA进行初始状态的闭包运算，在这之后，将新产生的状态子集作为新的DFA的初始状态，从该集合出发进行迭代和运算，首先将该状态子集中的状态进行遍历，与规定的字母表中的字母进行匹配运算，如果能够匹配到新的结果，便将该状态存入新产生的状态子集，但是由于会出现重复的情况，因此，在new_set生成之后，进行一步update()运算，其目的是将产生的新的状态子集中重复的状态进行删除。

```
void NFA::update(vector<string> &set1)
{
    if (set1.size() != 0)
    {
        for (int i = 0; i <= set1.size() - 1; i++)
        {
            for (int j = i; j < set1.size() - 1; j++)
            {
                if (set1[j].compare(set1[j + 1]) == 0)
                {
                    for (int k = j; k < set1.size() - 1; k++)
                    {
                        set1[k] = set1[k + 1];
                    }
```

```
                    set1.pop_back();
                }
            }
        }
    }
}
```

该部分进行更新运算，将集合中的元素进行遍历比较，出现相同时便进行移位然后清除。

在进行了该操作之后，就需要将新产生的状态子集进行保存，不过，在保存之前，首先需要保证已经产生的状态子集与新产生的这个子集没有出现重复，那么在保存之前需要进行一步检验操作，调用check()进行重复检验，如果出现重复的情况，就加一个标志位然后再保存进allset这一容器中。

```
bool NFA::check(vector<string> set1, vector<string> set2)
{
    int i;
    if (set1.size() != set2.size()||set1.size() == 0)
    {
        return false;
    }
    else
    {
        for (i = 0; i <= set1.size() - 1; i++)
        {
            if (set1[i].compare(set2[i])!= 0)
            {
                return false;
            }
        }
    }
    return true;
}
```

在产生了新的状态子集并且将其保存之后，就需要进行新的迭代，将相应的下标加1，以此来更改当前进行迭代的状态子集，但是由于集合中有一些是当时判断重复的子集，需要进行一步判断，状态子集不为空以及不含有"repeat"标志位的才继续进行迭代。当下标与allset的长度相等时，表示所有新产生的状态子集均已访问。

4.2.2 void rename() 新的状态进行重命名

```
void NFA::rename()
{
    for (int i = 0; i <= allset.size() - 1; i++)
    {
        if (allset[i].size() != 0 )
        {
            if (allset[i][allset[i].size() - 1].compare("repeat") != 0)
            {
                new_name.push_back(i);
            }
        }
    }
}
```

由于前面我们的状态子集的产生是按照一定的顺序去迭代和保存的，并且对于字母表中的所有字母来说，每一个状态都对应一个产生的状态子集，即使是空集也进行了保存，因此在进行状态子集的重命名时，我们直接进行遍历，将非空以及不是含“repeat”标志位的状态子集的下标保存在另一个new_name容器中，通过他们在该容器中的先后位置顺序进行重命名。

4.2.3 void output_dfa() 将产生的dfa进行输出

```
void NFA::output_dfa()
{
    int order1;
    int order2;
    int num = new_name.size();
    int dev = 1;
    for (int i = 0; i <= new_name.size() - 1; i++)
    {
        dev = (alphabet_num - 1) * i + 1;
        order1 = i + dev;
        for (int j = 0; j <= alphabet.size() - 1; j++)
        {
            order2 = findnum(allset[order1]);
            if (order2 != -1)
            {
                cout << i << " " << alphabet[j] << " " << order2 << endl;
            }
            order1 = order1 + 1;
        }
    }
}
```

通过前面的工作，可以看到，我们将所有DFA的状态保存在了new_name这一容器中，并且保存的是他们在原allset容器中的下标。最终DFA的输出形式仍然是以生成式的方式进行输出，可以看到，首先遍历整个状态集合，将每个状态进行一步字母表中状态的匹配，由于保存时每个字母对应的状态子集都保存，即使空集也保存，因此可以直接通过脚标与字母表个数之间的关系，直接计算出该状态对应的相应字母产生的状态子集的下标，然后通过findnum()函数，将该状态子集与new_name中的集合进行比较，如果相同，则将其在new_name中的顺序输出作为新的状态。

```
int NFA::findnum(vector<string> set1)
{
    int order;
    int num = 0;
    if (set1.size() != 0)
    {
        for (int i = 0; i <= new_name.size() - 1; i++)
        {
            order = new_name[i];
            if (set1[set1.size() - 1].compare("repeat") == 0)
            {
                if (set1.size() == (allset[order].size()+1))
                {
                    for (int j = 0; j < set1.size() - 1; j++)
                    {
                        if (set1[j].compare(allset[order][j]) == 0)
                        {
                            num++;
```

```
                        }
                    }
                    if (num == set1.size() - 1)
                    {
                        return i;
                    }
                    num = 0;
                }
            }
            else
            {
                if (check(set1, allset[order]))
                {
                    return i;
                }
            }
        }
    }
    return -1;
}
```

当然，在进行相应的下标查找时，也需要注意重复，以及长度为0也就是集合为空的情况。

4.2.4 void output_new_end() 输出新的终止状态

```
void NFA::output_new_end()
{
    int order;
    bool flag = true;
    for (int i = 0; i <= new_name.size() - 1 ; i++)
    {
        order = new_name[i];
        for (int j = 0; j <= allset[order].size() - 1 && flag; j++)
        {
            for (int k = 0; k <= end_set.size() - 1 && flag; k++)
            {
                if (allset[order][j].compare(end_set[k]) == 0)
                {
                    cout << i << " ";
                    flag = false;
                }
            }
        }
        flag = true;
    }
}
```

由于初始状态子集是由用户进行输入，并且allset也是从初始状态进行保存，更新，因此最终新的DFA初始状态一定是0，而对于新的终止状态则需要我们重新输出，而终止状态是那些新产生的状态子集中含有原先终止状态的子集，因此通过遍历new_name中的所有状态子集，并将其中的状态与原终止状态子集进行比较，如果有，则输出new_name中的下表。

五、程序的输入，输出，及测试

程序的输入：原NFA的生成式是通过文件进行输入，初始状态以及终止状态在程序中输入。

程序的输出：将新产生的DFA按照生成式的形式输出，并且将新的终止状态进行输出

测试：

```
4
0 a 1
1 a 1
1 a 2
2 b 1
2
```

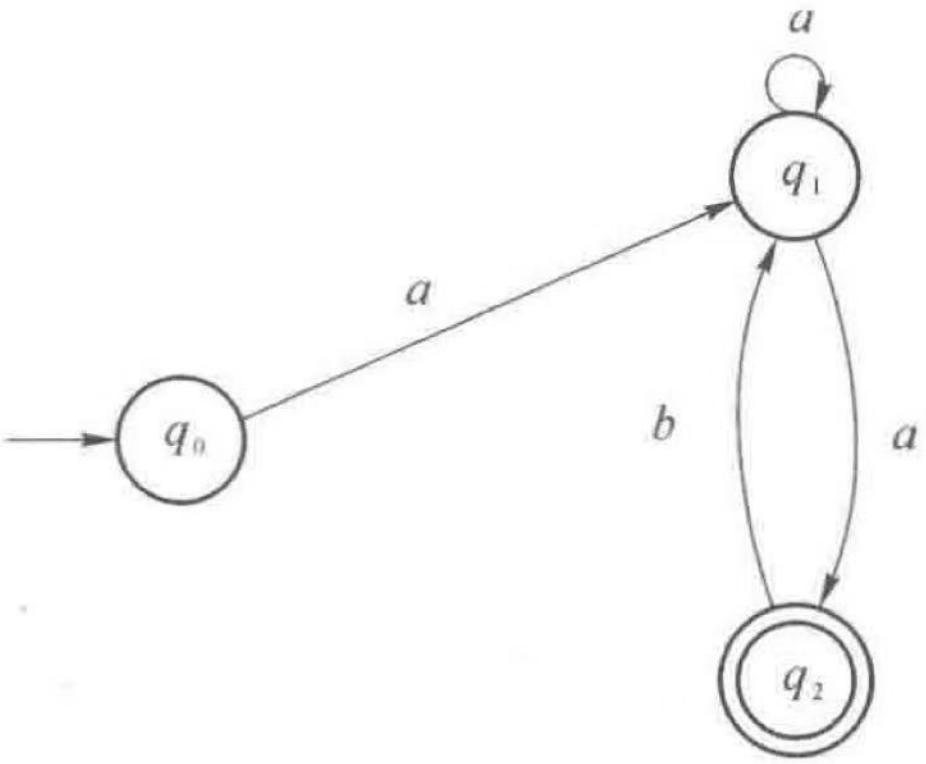
输入：

```
请输入你的起始状态个数：
1
请输入你的起始状态：
0
请输入你的终止状态个数：
1
请输入你的终止状态：
2
```

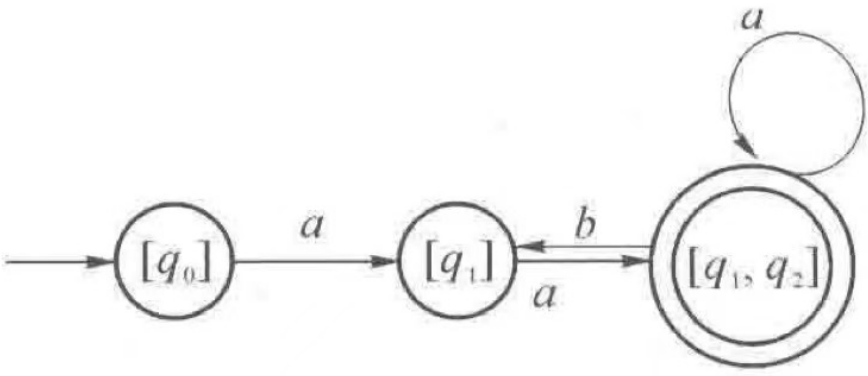
输出：

```
NFA:
0 a 1
1 a 1
1 a 2
2 b 1

DFA:
0 a 1
1 a 2
2 a 2
2 b 1

终止状态：
2
```

NFA:



DFA:



六、实验总结

通过本次实验，深入的理解了将NFA转换成DFA时的子集构造法的实现过程，通过将新的状态集合进行迭代是该算法的核心思想，并且在此过程中，团队成员分工明确，相互合作，将程序进行模块化编写，是我们每个人的团队开发能力得到了很大的提升，不仅如此，在此过程中，对于个人的编程能力也有很大的提高，可以将问题进行相应程度的抽象，使得每个人的逻辑思维能力得到了锻炼，这也是本次实验每位成员收获最大的地方。