形式语言与自动机——正则表达式执行器实验文档

一、实验环境

操作系统：windows10

cpu：intel i7

编译器：vs studio&vscode

编程语言：C++

二、主要实现思路

2.1 NFA的带路径执行

对于NFA的执行，我们注意到，由于其转义规则，每一条不同的路径都需要被遍历。因此，采用深度优先搜索的遍历方式，沿着一条路径一直执行，无法匹配则回溯寻找，便于执行。

对于输入字符与规则的判断，通过一个match函数，比对不同类型的规则与字符是否匹配来实现。

2.2 正则表达式构造自动机

采用递归的构造办法。我们知道，可以根据正则表达式生成一颗语法分析树。再分析树生成之后，我们考虑这样递归的形式构造自动机：对于一颗树根节点对应的自动机，可以由其孩子节点对应的自动机递归生成。

对于不同的根节点，子自动机合并的方式可能不同。例如，regex有子自动机的并组成，而expression由子自动机的交组成。

2.3 捕获分组的实现

对于每一个group，在其对应的自动机初态和终态添加分组标记。在合并过程中保留这些标记。

使得生成的自动机带有两个长为状态数的标记向量，分别标记该状态是否作为某一个分组的开始与结尾。

在NFA的DFS执行过程中，如果执行到某一状态，检查其开始与结尾向量中的元素并标记更新。

实例代码如下：

```
if(is_group[state])
    {
        for(int t=0;t<group_begins[state].size();t++)
        {
            id_group[group_begins[state][t]]=true;
            id_begins[group_begins[state][t]]=i;

        }
        for(int t=0;t<group_ends[state].size();t++)
```

```
        {
            id_group[group_ends[state][t]]=false;
            id_ends[group_ends[state][t]]=i;
        }
    }
```

2.4 anchor字符

改变NFA执行中match函数中epislon转移的规则。对于不同的anchor进行匹配即可

三、难点实现

3.1 NFA带路径执行中的epsilon转移自环问题

在执行过程中，如果出现epislon自环，则会造成DFS执行陷入死循环。

为了避免这一问题，采用一个二维数组存贮字符串执行位置i与状态t的遍历关系。保证对于每个状态，执行字符串索引为i的情型只出现一遍。

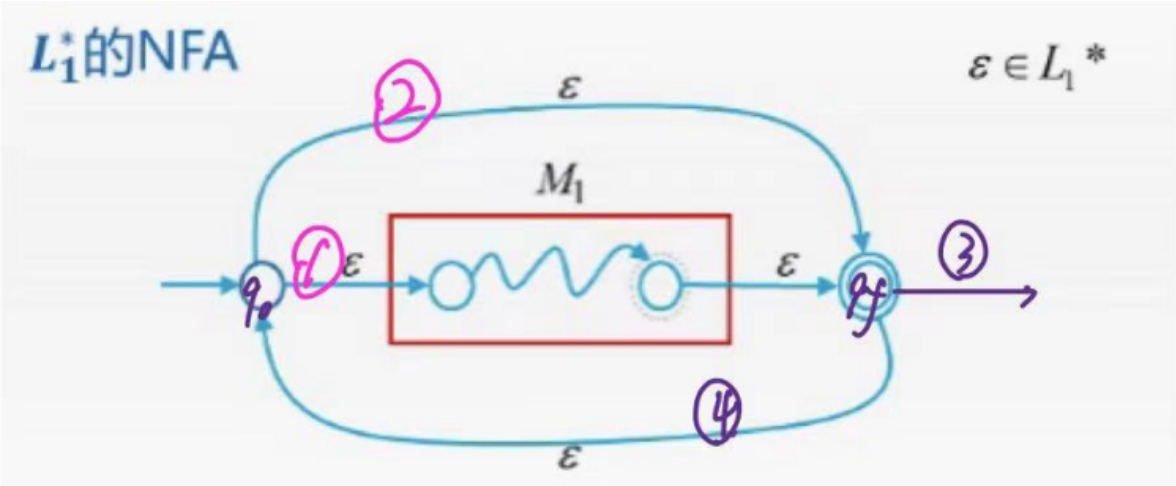
```
if (visited[rule.dst][i] == false)
            {
                ...
                visited[rule.dst][i] = true;
                if (DFS(text, rule.dst, a,init,end, i,
visited,id_begins,id_ends,id_group))
                    return true;
                ...
            }

        }
        else
        {
            if (visited[rule.dst][i + 1] == false)
            {
                if (i!=text.length()&&ObeytheRule(rule, text[i]))
                {
                    ...
                    visited[rule.dst][i+1] = true;
                    if (DFS(text, rule.dst, a, init ,end,i + 1,
visited,id_begins,id_ends,id_group))
                        return true;
                    ...
                }
            }
        }
```

3.2 quantifier的贪婪匹配问题

贪婪匹配是我第二次实验遇到的最大挑战。其原理简单却不容易思考，通过课件学习与相互讨论，我总结出了解决贪婪匹配的一般方法。

以最复杂的*闭包为例：



贪婪匹配即对于该自动机初态与终态转移规则的排序问题：

贪婪匹配要求q0的规则中，规则1在规则2之前，qf规则中，规则4在规则3之前。

非贪婪则反之。

3.3 character group的实现问题

注意到，character group包含多种不同类型的转移要求，同时还包括取反符。由于有取反符的存在，如果考虑构造交自动机较为复杂，于是考虑构造若干不同的转移规则来替代。注意到每一个要求都对应ASCII码0-127中的若干片段，我们通过二进制128位集合运算计算出最终要求能够实现的转移规则集合，即一个128位的二进制数。其中某一位位1表示支持该位对应的ASCII码符合要求。

我们取出最终二进制数中连续1的片段，每一个片段对应一个字符区间转移，即完成实现。

实例代码如下：

```
std::vector<int> convertToSet(const std::bitset<128>& set) {
    std::vector<int> result;
    bool isRange = false;
    int rangeStart = 0;

    for (int i = 0; i < 128; i++) {
        if (set[i]) {
            if (!isRange) {
                isRange = true;
                rangeStart = i;
            }
        } else {
            if (isRange) {
                isRange = false;
                result.push_back(rangeStart);
                result.push_back(i - 1);
            }
        }
```

```
    }

    if (isRange) {
        result.push_back(rangeStart);
        result.push_back(127);
    }

    return result;
}

std::bitset<128> convertCharToBinarySet(char input) {
    std::bitset<128> binarySet;

    int ascii = static_cast<int>(input);
    if (ascii >= 0 && ascii < 128) {
        binarySet.set(ascii, true);
    }

    return binarySet;
}

// 将字符区间的 ASCII 码转换为 128 位的二进制集合
std::bitset<128> convertStrToBinarySet(const std::string& input) {
    if (input.length() != 3 || input[1] != '-') {
        // 无效的输入格式
        return 0;
    }

    char startChar = input[0];
    char endChar = input[2];
    std::bitset<128> binarySet;

    for (int ascii = static_cast<int>(startChar); ascii <= static_cast<int>
(endChar); ascii++) {
        if (ascii >= 0 && ascii < 128) {
            binarySet.set(ascii, true);
        }
    }

    return binarySet;
}

// 将元字符转换为 129 位的二进制集合
std::bitset<128> convertMetaCharToBinarySet(const std::string& metaChar) {
    std::bitset<128> binarySet;

    char meta = metaChar[1];

    if (meta == 'd') {
        for (int i = 48; i <= 57; i++) {
            binarySet.set(i, true);
        }
    } else if (meta == 'D') {
        for (int i = 0; i < 48 || (i > 57 && i < 128); i++) {
            binarySet.set(i, true);
```

```
        }
    } else if (meta == 'w') {
        for (int i = 48; i <= 57; i++) {
            binarySet.set(i, true);
        }
        for (int i = 65; i <= 90; i++) {
            binarySet.set(i, true);
        }
        for (int i = 97; i <= 122; i++) {
            binarySet.set(i, true);
        }
        binarySet.set(95, true);
    } else if (meta == 'W') {
        for (int i = 0; i < 48 || (i > 57 && i < 65) || (i > 90 && i < 95) || (i
== 96) || (i > 122 && i < 128); i++) {
            binarySet.set(i, true);
        }
    } else if (meta == 's') {
        binarySet.set(' ', true);
        binarySet.set('\f', true);
        binarySet.set('\r', true);
        binarySet.set('\t', true);
        binarySet.set('\v', true);
    } else if (meta == 'S') {
        for (int i = 0; i < 128; i++) {
            if (i != ' ' && i != '\f' && i != '\r' && i != '\t' && i != '\v') {
                binarySet.set(i, true);
            }
        }
    } else if (meta == '.') {
        binarySet.set();
        binarySet.set('\n', false);
        binarySet.set('\r', false);

    }

    return binarySet;
}
```

3.5 分组匹配中的难点

在nfa中添加与状态类似的关于组别的4个向量，每个变量长度都为状态数，在涉及分组是时按照讲解的方式更新这些变量。

```
    std::vector<std::vector<int>> group_begins;
    std::vector<std::vector<int>> group_ends;
    int group_num=0;
    std::vector<bool> is_group;
```

在DFS的遍历过程中，如果一个状态对应组别标记，则进行更新组别的访问情况与开始结束位置。这种更新方式能保证每个组别记录的内容都是最后一次遍历的结果。

```
```

```
if(is_group[state])
    {
        for(int t=0;t<group_begins[state].size();t++)
        {
            id_group[group_begins[state][t]]=true;
            id_begins[group_begins[state][t]]=i;

        }
        for(int t=0;t<group_ends[state].size();t++)
        {
            id_group[group_ends[state][t]]=true;
            id_ends[group_ends[state][t]]=i;
        }
    }
```

注意到区间限定符同样需要返回最后一次遍历的组别内容。这里需要我们的思考：我们在进行类似aaa?a?的合并时，不改变组的序号。

在一般的合并a，b两个自动机时，b对应的组序号需要更具a中的组数提升相应的值。但此时我们不改变b中组序号，使得最后一次遍历的组别会被访问在正确的组别标记中。

这样的操作会使每个组别有多个开始结束标记。

3.6 matchall与replaceall

由于replaceall需要matchall执行过程中的下标信息，而我们有不能改变其返回值，因此，我们通过创建结构体和重构搜索逻辑的方式增添函数满足要求。

构造一个结构体用于返回match中匹配到的开始结尾索引

```
struct matchStr
{
    std::vector<std::string> matchstr;
    int init,end;
    bool if_match;
};
```

matchIndex函数返回matchstr

```
matchStr Regex::matchIndex(std::string text,int beginning)
```

match函数通过调用matchIndex函数，得到字符串向量。

matchAllIndex函数返回结构体向量

```
std::vector<matchStr> Regex::matchAllIndex(std::string text)
```

matchAll函数调用matchAllIndex函数，得到目标向量。

ReplaceAll函数调用MatchAllIndex函数，得到每一个匹配的开始结束索引与组别信息，实现功能。

四、感悟

这次自动机大作业，在特别忙碌的大二下学期，是一次非常大的挑战。这三次实验的要求层层递进，其中要求的许多功能也向我提出了不小的挑战。与此同时，在编程过程中，还需要注重代码的美观整洁，增强其面向对象的属性，也常常让我把一段代码重写上三四遍。

然而，当最后一个样例通过的一刻，我心中油然而生一种成就感。经过一个学期的努力，我成功完成了一项大工程，不仅替升了对于形式语言自动机的理解，也让我在C++编程，debug，数学逻辑等多方面有了显著的提高！

虽路途艰辛，但终尝所愿！