**北京邮电大学《形式语言与自动机》课程实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **实验**  **名称** | NFA到DFA的转化 | | **学 院** | 计算机 | **指导教师** | 邵苏杰 |
| **班 级** | **学 号** | **学生姓名** | | **分工** | | |
| **2023211311** | **2023211196** | **马一民** | | 负责编写输入输出，实现I/O的多样化。同时设计具体的测试用例，对程序进行全面的测试，检查程序的正确性和稳定性，并协助进行代码调试。同时，负责实验报告的完善工作。 | | |
| **2023211311** | **2023211202** | **张鑫溢** | | 主要负责NFA 到 DFA 转化核心算法的设计与实现，承担算法类代码的编写任务，并对程序进行初步测试且撰写部分实验报告。 | | |
| **2023211311** | **2023211209** | **肖璨** | | 负责 ε-NFA 、NFA 、DFA 数据结构的设计与实现，包括自动机与文法格式的定义和解析，同时参与实验报告撰写和部分代码编写工作。 | | |
| **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。 | | | | | |
| **课**  **程**  **设**  **计**  **成**  **绩**  **评**  **定** | **评语**:  **成绩**:  指导教师签名：  年 月 日 | | | | | |

注：评语要体现每个学生的工作情况，可以加页。

1. **实验目的**

编程实现将不确定有限自动机（NFA）转化为确定有限自动机（DFA）的算法，深入理解和掌握不同类型有限状态自动机之间的转化原理与过程。通过实际的编程操作，能够熟练运用相关知识解决自动机转化的问题，并且在团队协作中提升分析问题、解决问题以及沟通协作的能力，为后续在实际应用中利用自动机理论刻画和处理系统相关问题奠定基础。

**二、实验环境**

1. 编程语言：Python，版本 Python 3.11 ，利用其简洁的语法和丰富的标准库，方便进行算法实现和数据处理。

2. 开发工具：使用 Visual Studio Code（vscode）作为代码编辑器，其具有丰富的插件生态系统，支持 Python 代码的语法高亮、代码补全、调试等功能。同时使用Graphviz对自动机实现可视化输出。

3. 版本控制：采用 Git 作为版本控制系统，通过在 Git 仓库（GitHub上的私有仓库）中进行代码交互，方便团队成员之间对代码的版本管理、协作开发以及代码合并等操作。

4. 运行环境：操作系统为 windows 11，在该环境下安装了 Python 运行时环境以及相关依赖库，以确保代码能够正确运行。

1. **实验内容**

有限状态自动机是描述控制过程的有力工具，在计算机科学领域有着广泛的应用，例如编译器的词法分析、模式匹配等。有限状态自动机主要有确定有限状态自动机（DFA）和不确定有限状态自动机（NFA）两种类型。这两种类型的自动机在表达能力上是等价的，即可以通过一定的算法将 NFA 转化为与之等价的 DFA。

在实际应用中，有时使用 NFA 来刻画实际系统更加方便，因为它在状态转移的描述上更加灵活，允许在同一状态下对于相同的输入字符有多个转移选择。而 DFA 在处理和实现上相对更加简单和高效，因此需要掌握 NFA 到 DFA 的转化方法。

本实验的具体任务是通过编程实现 NFA 到 DFA 的自动转化。要求自行设定不确定有限自动机的描述格式作为输入，经过算法处理后输出对应的确定有限自动机。在编程过程中，需要综合运用自动机的相关理论知识，设计合理的算法和数据结构来完成转化过程，并通过测试用例对程序的正确性进行验证。

1. **程序的设计思路及核心算法**

1、设计思路：

首先，定义 NFA 和 DFA 的数据结构。对于 NFA，使用字典来表示状态转移函数，键为状态和输入字符的元组，值为可能转移到的状态集合。同时记录 NFA 的起始状态和接受状态集合。

对于 DFA，同样使用字典表示状态转移函数，不过每个状态对于每个输入字符只有一个确定的转移状态。另外，维护 DFA 的状态集合、起始状态和接受状态集合。

从 NFA 的起始状态开始，计算其 ε 闭包（即从该状态通过一系列 ε 转移能够到达的所有状态的集合），将这个 ε 闭包作为 DFA 的起始状态。

对于 DFA 的每个状态（初始为起始状态），针对每个输入字符，计算从该状态出发通过该输入字符能够到达的状态集合（先通过 NFA 的状态转移函数找到可能的转移状态，再计算这些状态的 ε 闭包），将这个新的状态集合作为 DFA 中该状态对于该输入字符的转移状态。如果这个新的状态集合尚未在 DFA 的状态集合中出现，则将其添加到 DFA 的状态集合中。

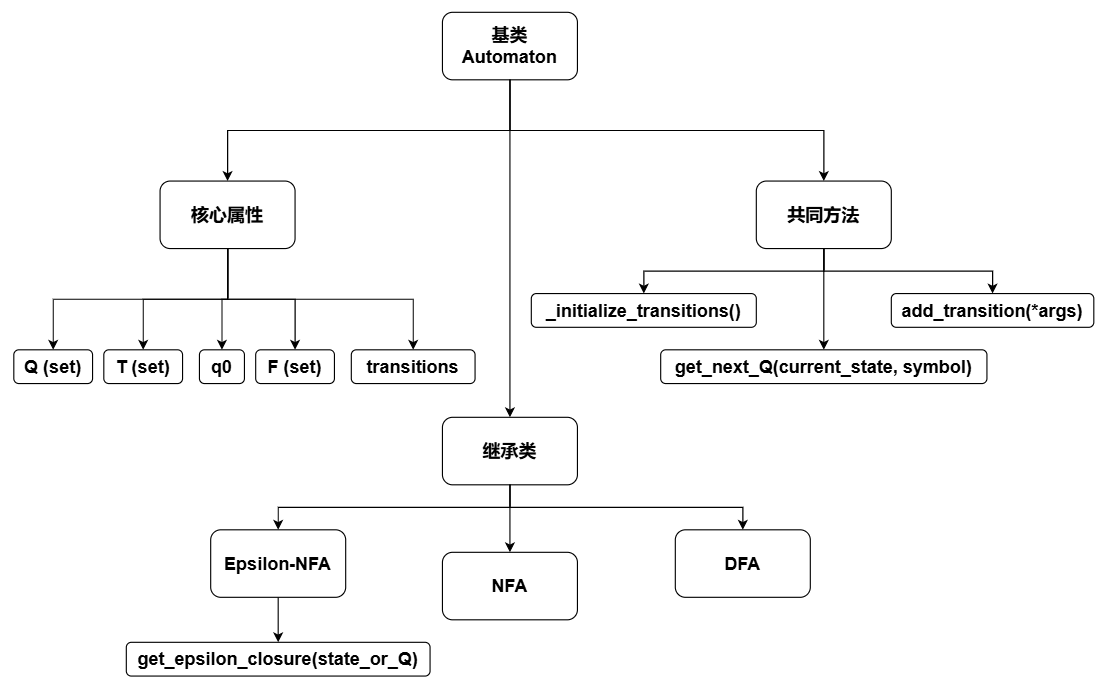
重复上述步骤，直到 DFA 的状态集合不再增加为止。

确定 DFA 的接受状态集合，即包含 NFA 接受状态的 DFA 状态集合中的状态。

2、数据结构设计

2.1. 整体设计思路

本次实验采用面向对象编程的设计思想来构建自动机的数据结构。核心思路是定义一个抽象基类 Automaton，它包含了所有类型自动机（ε-NFA, NFA, DFA）共有的属性和行为接口。然后，通过继承机制，分别实现具体的 Epsilon-NFA、NFA 和 DFA 类，它们继承基类的共同特性，并各自实现或重写特定的方法以满足自身的定义。



2.2. 抽象基类 Automaton

作为所有自动机类型的统一模板，Automaton类定义了自动机的基本组成部分和必须实现的操作。

实现: 使用 Python 的 abc 模块，继承自 abc.ABC。

|  |  |  |
| --- | --- | --- |
|  | 变量与函数 | 功能 |
| 核心属性 | **Q (set)** | 存储自动机的状态集合。 |
| **T (set)** | 存储输入字母表。 |
| **q0** | 存储起始状态。 |
| **F (set)** | 存储接受状态集合。 |
| **transitions** | 存储转移函数 |
| 抽象方法 | **\_initialize\_transitions()** | 初始化转移函数 |
| **add\_transition(\*args)** | 添加转移规则 |
| **get\_next\_Q(current\_state, symbol)** | 查找下一状态（或状态集）的方法 |
| 具体方法 | **\_\_init\_\_(self, Q, T, q0, F)** | 初始化所有自动机共有的属性 (Q, T, q0, F)， |
| **\_\_repr\_\_(self)** | 基本属性显示 |

**核心属性:**

**Q (set):** 存储自动机的状态集合。

**T (set):** 存储输入字母表。

**q0:** 存储起始状态。

**F (set):** 存储接受状态集合。

**transitions:** 存储转移函数。其具体结构由子类在 \_initialize\_transitions 方法中定义和初始化。

**抽象方法 (@abc.abstractmethod):**

**\_initialize\_transitions():** 要求子类提供初始化转移函数数据结构的方法。

**add\_transition(\*args):** 要求子类提供添加转移规则的具体实现。

**get\_next\_Q(current\_state, symbol):** 要求子类提供根据当前状态和输入符号查找下一状态（或状态集）的方法。

**具体方法:**

**\_\_init\_\_(self, Q, T, q0, F):**

初始化所有自动机共有的属性 (Q, T, q0, F)，并调用 \_initialize\_transitions 初始化转移函数。包含对输入类型的基本检查。对 q0 和 F 的成员资格检查有所放宽，以适应 NFA 到 DFA 转换过程中状态集合动态变化的特性。

**\_\_repr\_\_(self):**

提供一个通用的字符串表示形式，显示基本属性，但转移函数的具体内容留给子类实现。

2.3. 自动机继承类

2.3.1. Epsilon-NFA 类

继承自 Automaton，带有 Epsilon (ε) 转移的非确定性有限自动机。

|  |  |  |
| --- | --- | --- |
|  | 变量与函数 | 功能 |
| 常量定义 | **EPSILON = ''** | Epsilon 符号定义 |
| 额外数据结构 | **transitions (dict)** | 具体实现的转移函数 |
| 具体方法实现 | **\_initialize\_transitions()** | 转移函数初始化 |
| **add\_transition(from\_state, symbol, to\_Q)** | 添加NFA转移逻辑 |
| **get\_next\_Q(current\_state, symbol)** | 查找下一个转移 |
| **get\_epsilon\_closure(state\_or\_Q)** | 计算单个状态或状态集合的 Epsilon 闭包。 |
| **\_\_repr\_\_()** | 提供更详细的输出。 |

**常量定义:**

EPSILON = '': 定义空字符串作为 Epsilon 符号。

**数据结构:**

**transitions (dict):**

实现为字典。键是 (state, symbol) 的元组，其中 symbol 可以是字母表中的符号或 EPSILON。值是一个 set，包含所有可能的下一个状态。

**方法实现:**

**\_initialize\_transitions():** 返回一个空字典 {}。

**add\_transition(from\_state, symbol, to\_Q):** 实现添加转移规则的逻辑。

如果键 (from\_state, symbol) 不存在，则创建新条目并初始化为空集合；然后使用 set.update() 将 to\_Q 中的状态添加到目标集合中。同时，确保 from\_state 和 to\_Q 中的所有状态都被添加到 self.Q 集合中。包含对输入状态、符号和目标状态集合类型的检查。

**get\_next\_Q(current\_state, symbol):**

使用 dict.get(key, default) 方法查找转移。如果找到 (current\_state, symbol) 的转移，返回对应的状态集合；否则返回一个空集合 set()。

**get\_epsilon\_closure(state\_or\_Q):**

计算单个状态或状态集合的 Epsilon 闭包。采用BFS遍历（队列实现）的方式，查找所有仅通过 ε 转移可达的状态集合。使用 processed 集合防止无限循环。

**\_\_repr\_\_():**

重写基类方法，提供更详细的输出。对状态集、字母表、接受状态集进行排序显示。特别地，它会格式化并排序 transitions 字典的内容，使输出更清晰、一致。

2.3.2. NFA 类

继承自 Epsilon-NFA（已经包含了处理状态集合作为转移目标的大部分逻辑），标准的非确定性有限自动机（不允许 Epsilon 转移）。

**关键差异:**

**add\_transition(from\_state, symbol, to\_Q):**

重写以禁止添加 Epsilon 转移。如果 symbol 是 EPSILON，则抛出 ValueError。对于非 Epsilon 转移，调用 super().add\_transition() 复用 Epsilon-NFA 的添加逻辑。

**get\_epsilon\_closure(state\_or\_Q):**

由于标准 NFA 没有 Epsilon 转移，任何状态或状态集的 Epsilon 闭包就是其自身。因此，该方法直接返回输入状态（包装成集合）或输入集合的副本。

**\_\_repr\_\_:**

继承自 Epsilon-NFA，对没有 Epsilon 转移的 NFA 同样适用。

2.3.3 DFA 类

继承自 Automaton的确定性有限自动机。

|  |  |  |
| --- | --- | --- |
|  | 变量与函数 | 功能 |
| 额外数据结构 | **transitions (dict)** | 具体实现的转移函数 |
| 具体方法实现 | **\_initialize\_transitions()** | 转移函数初始化 |
| **add\_transition(from\_state, symbol, to\_Q)** | 添加DFA转移逻辑 |
| **get\_next\_Q(current\_state, symbol)** | 查找下一个转移 |
| **is\_complete** | DFA完整性检查 |
| **\_\_repr\_\_()** | 提供DFA具体输出 |

**数据结构:**

**transitions (dict):**

与NFA 不同，值是单个下一状态。在 NFA 到 DFA 的转换过程中， DFA 的状态通常是 NFA 状态的集合。

**方法实现:**

**\_initialize\_transitions():** 返回一个空字典 {}。

**add\_transition(from\_state, symbol, to\_state):**

实现添加 DFA 转移规则。关键在于检查确定性：如果键 (from\_state, symbol) 已存在且其对应的 to\_state 与新传入的不同，则抛出 ValueError，因为 DFA 不允许非确定性转移。同时，确保 from\_state 和 to\_state 都被添加到 self.Q 集合中。

**get\_next\_Q(current\_state, symbol):**

使用 dict.get(key) 方法查找转移。如果找到，返回单个下一状态；否则返回 None（表示该转移未定义）。

**is\_complete():**

提供一个辅助方法来检查 DFA 是否是完整的（即是否为每个状态和每个字母表符号都定义了转移）。

**\_\_repr\_\_():**

重写基类方法。包含一个内部辅助函数 state\_to\_str 来优雅地处理可能为 frozenset 的状态，将其显示为更易读的集合形式（如 '{0, 1}'）。对状态、接受状态和转移进行排序和格式化输出。

3、转换算法

根据定义的NFA和epsilon-NFA类以及DFA类，将转换的思路拆分为将epsilon-NFA转换为NFA和NFA转换为DFA。

3.1. epsilon-NFA转换为NFA

to\_nfa 函数实现ε转移的非确定性有限自动机（ε-NFA）转换为等价的不含ε转移的 NFA，其核心逻辑围绕着对ε闭包的计算与运用，以此来消除ε转移。

to\_nfa函数首先创建一个全新的NFA对象，该对象将继承原 ε-NFA 的字母表与初始状态，不过此时的状态集合和接受状态集合均为空，后续会在处理过程中逐步填充。

接下来处理状态转移。对于原epsilon-NFA中的每一个状态，算法都会计算其ε闭包，即通过任意多次ε转移能够到达的所有状态。

在ε闭包计算的基础上后，设是一个，对应的为，对于每个状态 s 和每个输入符号 a，根据来确定新的转移函数。针对字母表中的每一个符号，先找出通过该符号直接可达的状态集合；紧接着，对这些直接可达的状态再次计算其闭包，并将得到的结果合并到next\_states集合中。这样一来，next\_states集合就涵盖了从当前状态通过特定符号并包含ε转移后能够到达的所有状态。若next\_states集合不为空，便将其作为转移规则添加到新构建的NFA里。

同时要对接受状态进行判定。若某个状态的ε闭包包含原epsilon-NFA的任何一个接受状态，那么该状态就会被标记为新NFA的接受状态。

最后，为确保新NFA的完整性，算法会将原epsilon-NFA中的所有状态都添加到新NFA的状态集合中，至此，整个epsilon-NFA到NFA的转换过程完成，最终返回构建好的不含 ε 转移的 NFA。

具体代码如下

    def to\_nfa(self):

        """将ε-NFA转换为等价的NFA。"""

        # 创建新的NFA

        nfa = NFA(set(), self.T.copy(), self.q0, set())

        # 对于每个状态和每个字母表符号，计算包含ε转移的转移

        for state in self.Q:

            # 计算状态的ε闭包

            epsilon\_closure = self.get\_epsilon\_closure(state)

            # 对于字母表中的每个符号

            for symbol in self.T:

                next\_states = set()

                # 对于ε闭包中的每个状态，找到它们通过symbol可以到达的状态

                for epsilon\_state in epsilon\_closure:

                    direct\_states = self.get\_next\_Q(epsilon\_state, symbol)

                    # 对于每个直接到达的状态，加入它的ε闭包

                    for direct\_state in direct\_states:

                        next\_states.update(self.get\_epsilon\_closure(direct\_state))

                # 如果有可到达的状态，添加到NFA中

                if next\_states:

                    nfa.add\_transition(state, symbol, next\_states)

            # 如果状态的ε闭包包含任何接受状态，则该状态在NFA中也是接受状态

            if any(s in self.F for s in epsilon\_closure):

                nfa.F.add(state)

        # 确保所有状态都被添加到NFA中

        nfa.Q.update(self.Q)

        return nfa

3.2. NFA转换为DFA

在不含ε转移的nfa基础上，to\_dfa函数采用“子集构造法”，即用 NFA 状态的集合来表示 DFA 的单个状态，实现了NFA到DFA的转换。

首先，创建一个新的DFA对象，此时它的状态集合为空，字母表沿用NFA的字母表，初始状态已确定，接受状态集合也为空。

先确定DFA的初始状态，这个初始状态是通过计算NFA初始状态的ε闭包得到的，它是一个由NFA初始状态构成的集合。同时，创建用于存储DFA状态的集合、记录DFA接受状态的集合，以及存放待处理DFA状态的队列，并将初始状态加入队列和状态集合中。随后，

进入状态处理循环后，算法从待处理队列中取出当前DFA状态。针对字母表中的每一个输入符号，开始计算通过该符号从当前DFA状态（NFA状态子集）能够到达的所有NFA状态。当存在可到达的NFA状态时，将这些状态组合成新的DFA状态，并添加到DFA的转移规则中；若该新DFA状态尚未出现过，就将其加入到DFA的状态集合和待处理队列，以便后续进一步处理。若针对某个输入符号不存在可到达的状态，算法会引入一个空集状态（即死状态），并将当前状态对应该符号的转移指向死状态。若死状态是新出现的，还需为其定义所有输入符号的自循环转移，确保DFA状态转移的完整性。

当所有DFA状态处理完毕，需要确立接受状态。遍历DFA的每一个状态，检查其中是否包含NFA的接受状态，若包含，则将该DFA状态标记为接受状态。最后，更新DFA的状态集合，使其完整包含所有处理过的状态，至此，NFA 到DFA的转换结束，返回构建好的 DFA。

具体代码如下

    def to\_dfa(self):

        """将NFA转换为等价的DFA。"""

        # 创建DFA的初始状态集

        dfa\_initial = frozenset({self.q0})

        dfa\_states = {dfa\_initial}  # 存储DFA状态（每个状态是NFA状态的一个子集）

        dfa\_accept = set()  # DFA的接受状态

        unprocessed = [dfa\_initial]  # 待处理的DFA状态队列

        # 创建一个新的DFA

        dfa = DFA(set(), self.T.copy(), dfa\_initial, set())

        # 处理所有DFA状态

        while unprocessed:

            current\_dfa\_state = unprocessed.pop(0)

            # 对于每个输入符号

            for symbol in self.T:

                next\_nfa\_states = set()

                # 计算通过当前符号可以到达的所有NFA状态

                for nfa\_state in current\_dfa\_state:

                    next\_nfa\_states.update(self.get\_next\_Q(nfa\_state, symbol))

                # 如果有下一个状态，创建新的DFA转移

                if next\_nfa\_states:

                    # 将NFA状态集转换为DFA状态（frozenset）

                    next\_dfa\_state = frozenset(next\_nfa\_states)

                    # 添加转移

                    dfa.add\_transition(current\_dfa\_state, symbol, next\_dfa\_state)

                    # 如果这是一个新的DFA状态，添加到待处理列表

                    if next\_dfa\_state not in dfa\_states:

                        dfa\_states.add(next\_dfa\_state)

                        unprocessed.append(next\_dfa\_state)

                else:

                    # 没有下一状态，转移到空集状态（死状态）

                    empty\_state = frozenset()

                    dfa.add\_transition(current\_dfa\_state, symbol, empty\_state)

                    # 如果死状态是新状态，添加到状态集

                    if empty\_state not in dfa\_states:

                        dfa\_states.add(empty\_state)

                        # 处理死状态的转移

                        for sym in self.T:

                            dfa.add\_transition(empty\_state, sym, empty\_state)

        # 确定DFA的接受状态

        for dfa\_state in dfa\_states:

            # 如果DFA状态包含任何NFA接受状态，则它是接受状态

            if any(nfa\_state in self.F for nfa\_state in dfa\_state):

                dfa.F.add(dfa\_state)

        # 更新DFA的状态集

        dfa.Q = dfa\_states

        return dfa

4、I/O设计

本实验的输入输出设计采用交互式命令行界面，使用两级目录来选择实现对应操作。同时本次实验以一定的格式输入输出自动机，实现了自动机的文本和图形化输出。减少重复输入自动机带来的繁琐操作，实现了自动机的保存读取功能，具体设计如下：

首先一级菜单中实现了三种对应类型自动机的创建，包含直接输入创建、从文件加载两种方法。直接输入创建从包含状态集合、包含字母集合、状态转移、初始状态、接受状态五个方面输入创建一个自动机；从文件加载需要对文件的格式有一定要求。两种方式相互补充，实现了自动机的输入。

下面是用户输入的示例：

=== 创建 EpsilonNFA ===

输入状态集 (用逗号分隔，如: q0,q1,q2): q0,q1,q2,q3,q4,q5

输入字母表 (用逗号分隔，如: a,b,c): +,-,0,.

输入初始状态: q0

输入接受状态集 (用逗号分隔): q5

添加转移 (输入空行结束):

格式: 当前状态,输入符号,目标状态1,目标状态2,... (对于ε转移，输入符号留空)

转移: q0,,q1

已添加转移: (q0, ε) -> {'q1'}

转移: q0,+,q1

已添加转移: (q0, +) -> {'q1'}

转移: q0,-,q1

已添加转移: (q0, -) -> {'q1'}

转移: q1,0,q1

已添加转移: (q1, 0) -> {'q1'}

转移: q1,.,q2

已添加转移: (q1, .) -> {'q2'}

转移: q1,0,q4

已添加转移: (q1, 0) -> {'q4'}

转移: q2,0,q3

已添加转移: (q2, 0) -> {'q3'}

转移: q4,.,q3

已添加转移: (q4, .) -> {'q3'}

转移: q3,0,q3

已添加转移: (q3, 0) -> {'q3'}

转移: q3,,q5

已添加转移: (q3, ε) -> {'q5'}

转移:

EpsilonNFA 创建成功:

EpsilonNFA(

Q=['q0', 'q1', 'q2', 'q3', 'q4', 'q5'],

T=['+', '-', '.', '0'],

transitions={

(q0, ε): {'q1'},

(q0, '+'): {'q1'},

(q0, '-'): {'q1'},

(q1, '.'): {'q2'},

(q1, '0'): {'q1', 'q4'},

(q2, '0'): {'q3'},

(q3, ε): {'q5'},

(q3, '0'): {'q3'},

(q4, '.'): {'q3'},

},

q0=q0,

F=['q5']

)

在二级菜单中，对于当前的自动机提供了查看当前自动机、验证当前自动机，以及核心的自动机的转换功能，并且实现了自动机的文本和可视化输出。

示例输出如下：

转换完成，生成的DFA:

DFA(

Q={{'q0'}, {'q1', 'q4'}, {'q1'}, {'q2'}, {'q3', 'q5'}, {'q5', 'q3', 'q2'}, {}},

T=['+', '-', '.', '0'],

transitions={

({'q0'}, '+'): {'q1'},

({'q0'}, '-'): {'q1'},

({'q0'}, '.'): {'q2'},

({'q0'}, '0'): {'q1', 'q4'},

({'q1', 'q4'}, '+'): {},

({'q1', 'q4'}, '-'): {},

({'q1', 'q4'}, '.'): {'q5', 'q3', 'q2'},

({'q1', 'q4'}, '0'): {'q1', 'q4'},

({'q1'}, '+'): {},

({'q1'}, '-'): {},

({'q1'}, '.'): {'q2'},

({'q1'}, '0'): {'q1', 'q4'},

({'q2'}, '+'): {},

({'q2'}, '-'): {},

({'q2'}, '.'): {},

({'q2'}, '0'): {'q3', 'q5'},

({'q3', 'q5'}, '+'): {},

({'q3', 'q5'}, '-'): {},

({'q3', 'q5'}, '.'): {},

({'q3', 'q5'}, '0'): {'q3', 'q5'},

({'q5', 'q3', 'q2'}, '+'): {},

({'q5', 'q3', 'q2'}, '-'): {},

({'q5', 'q3', 'q2'}, '.'): {},

({'q5', 'q3', 'q2'}, '0'): {'q3', 'q5'},

({}, '+'): {},

({}, '-'): {},

({}, '.'): {},

({}, '0'): {},

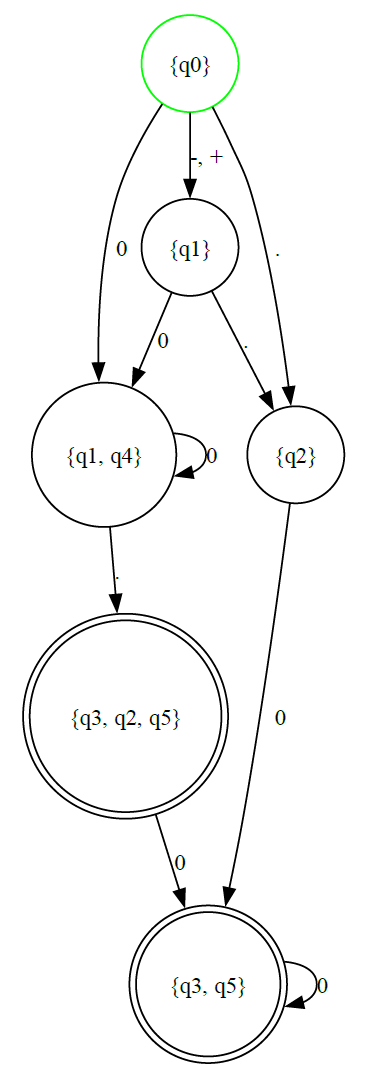
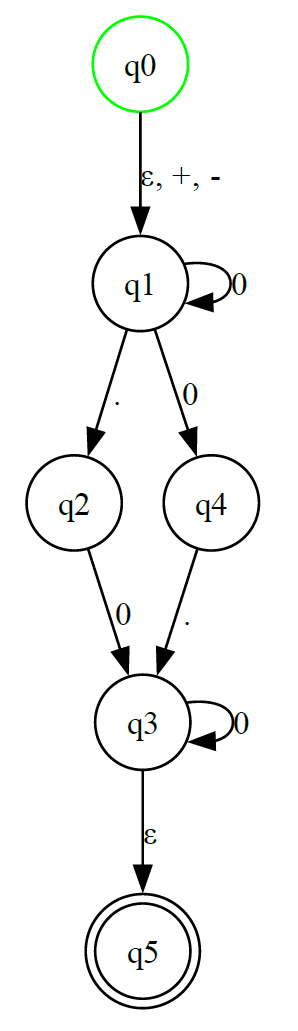
},

q0={'q0'},

F={{'q3', 'q5'}, {'q5', 'q3', 'q2'}}

)

自动机的可视化输出主要通过成熟完整的Graphviz库进行实现。对于三种不同的状态机分为NFA和DFA两种格式的可视化输出，主要依赖于Graphviz库实现：将当前能够实现的自动机的状态转移的描述利用Grahviz库所提供的不同图形转化成可视化自动机，其中特别要注意的是初始状态用绿色圆圈标出、接受状态用两层圆圈表示。

可视化输出时，若输入文件名称则会对应生成Graphviz支持的代码文件以及可视化图形的pdf文件；若没有输入，则直接展示可视化图形。

以上两图分别为NFA、DFA的可视化图形。

1. **实验总结**

本次实验通过编程实现了NFA到DFA的自动转换，深入理解了有限自动机的基本原理及其等价转化过程。实验过程中，我们设计并实现了支持Epsilon-NFA、NFA和DFA的自动机类，采用子集构造法等经典算法，实现了自动机的高效转换。通过交互式命令行界面，用户可以方便地输入自动机、测试字符串、保存和加载自动机，并对自动机进行可视化展示。

实验不仅加深了对自动机理论的理解，也锻炼了将理论算法转化为实际代码的能力。在调试和测试过程中，我们体会到自动机状态管理、转移函数设计和输入输出交互等细节对程序正确性和用户体验的重要性。通过本实验，我们掌握了自动机等价转化的基本方法，为后续深入学习形式语言与自动机理论及其应用打下了坚实基础。