OT1 设计报告

从NFA到DFA的自动机转换

数据结构

我们使用有向图结构来表示DFA，图中节点为DFA的状态，边为转换条件。

图结构主要包含以下几种方法：

1. `get_edge(self, start, end)`: 用于查找给定起始节点 `start` 和终止节点 `end` 之间的边。它遍历 `start` 节点的出边列表，并返回与 `end` 节点相关的边的值。如果没有这样的边，它返回0作为标志。
2. `get_end(self, start, val)`: 用于查找从给定起始节点 `start` 出发，标记为 `val` 的所有终止节点。它遍历 `start` 节点的出边列表，查找与 `val` 标记匹配的边，然后返回相应的终止节点列表。
3. `add_edge(self, start, end, value)`: 它接受 `start`（起始节点）、`end`（终止节点）和 `value`（边的值）作为参数，并将这些信息添加到 `edges` 数据结构中，以表示从 `start` 到 `end` 的边。

处理状态转移

我们设计了三个函数来处理有限自动机的状态转移和ε-闭包计算。

1. `get_key(dic, val)`: 此函数用于根据字典 `dic` 中的值 `val` 查找对应的键。它返回一个包含所有匹配的键的列表。
2. `move(graph, T, val)`: 此函数接受一个有向图 `graph`、状态集合 `T` 和一个值 `val`，并返回状态集合 `T` 中所有满足从任一状态经过值 `val` 可达的状态的集合。
3. `eps_cover(graph, T)`: 这是一个递归函数，它计算状态集合 `T` 的ε-闭包。ε-闭包包含了从状态集合 `T` 出发，经过任意数量的ε-转移可达的状态。函数首先将状态集合 `T` 的每个状态添加到结果集中，然后递归地查找从当前状态通过ε-转移可达的其他状态并添加到结果中。

核心思路

在`main`函数中我们使用`state_name` 存储了 DFA 的状态标识符，`state_change` 存储了从一个 DFA 状态经过输入符号到达下一个 DFA 状态的转移关系。

```
    while 1:
        curr_state = wait[i]
        name += 1
        state_name[chr(name)] = curr_state
        father = chr(name)
        change = []

        a = eps_cover(graph, move(graph, curr_state, 'a'))
        change.append(a)

        b = eps_cover(graph, move(graph, curr_state, 'b'))
        change.append(b)

        if a not in wait:
            wait.append(a)
            total += 1
        if b not in wait:
            wait.append(b)
            total += 1

        i += 1
        state_change[father] = change

        if total < i:
            break
```

程序的主要循环部分：

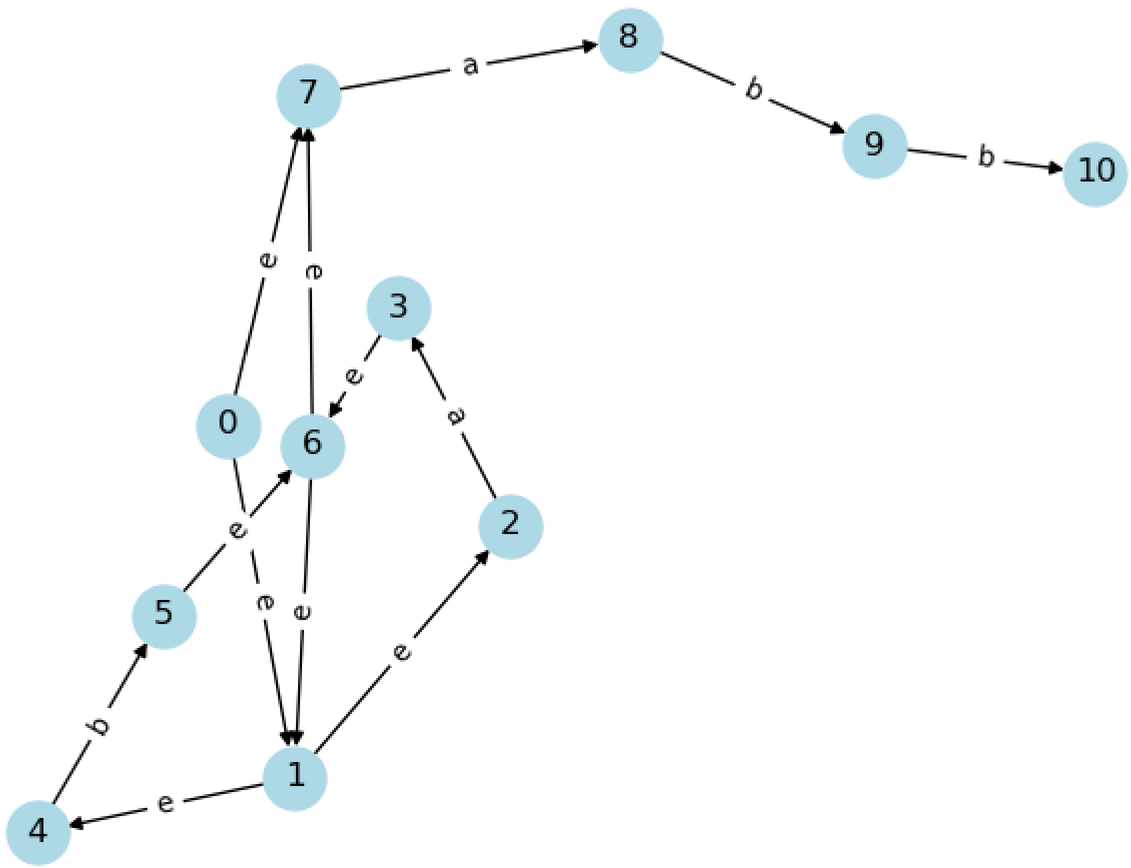
- 程序首先将初始状态集合 `[0]` 的 ε-闭包添加到 `wait` 队列中。
- 然后进入一个无限循环，直到完成 DFA 转换。
- 在每次循环迭代中，程序从 `wait` 队列中取出当前状态 `curr_state`。
- 针对每个输入符号 'a' 和 'b'，程序计算通过 ε-闭包得到的新状态集合，并将它们存储在 `change` 中。
- 如果这些新状态集合不在 `wait` 中，它们被添加到 `wait` 队列，并递增 `total`。

- 程序使用一个唯一的字符（例如 'A'、'B'）来表示当前状态，并将这个状态与相应的状态集合建立映射。
- 最后，程序将从当前状态通过输入符号 'a' 和 'b' 得到的新状态集合存储在 `state_change` 中，并继续下一个循环迭代。
- 当 `total` 不再增加，即所有可能的状态都已处理时，程序退出循环，DFA 构造完成。

最后我打印出DFA结果并调用draw函数绘制DFA图。

运行结果

NFA：



DFA：

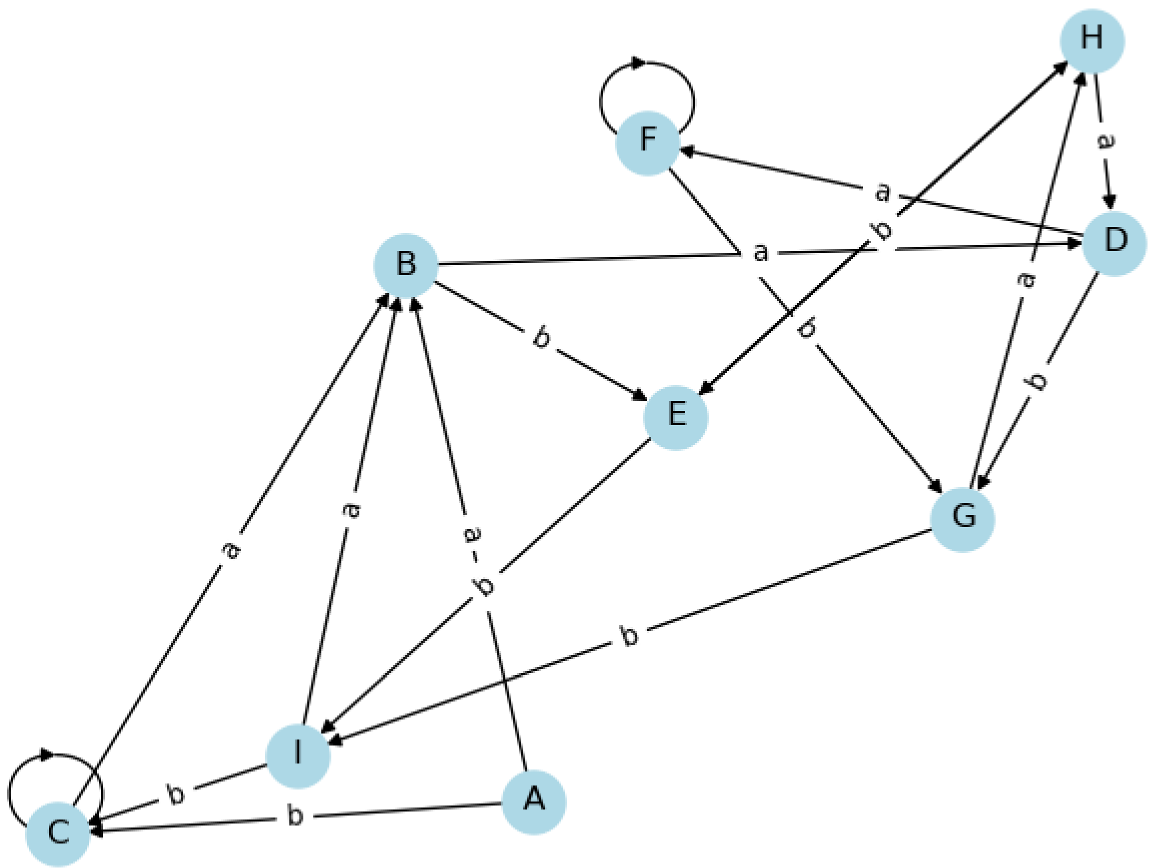
```
A [0, 1, 2, 4, 7]
B [1, 2, 3, 4, 6, 7, 8]
C [1, 2, 4, 5, 6, 7]
D [1, 2, 3, 4, 6, 7, 8, 9]
E [1, 2, 4, 5, 6, 7, 9]
```

```
F [1, 2, 3, 4, 6, 7, 8, 9, 10]
G [1, 2, 4, 5, 6, 7, 9, 10]
H [1, 2, 3, 4, 6, 7, 8, 10]
I [1, 2, 4, 5, 6, 7, 10]




A {'a': 'B', 'b': 'C'}
B {'a': 'D', 'b': 'E'}
C {'a': 'B', 'b': 'C'}
D {'a': 'F', 'b': 'G'}
E {'a': 'H', 'b': 'I'}
F {'a': 'F', 'b': 'G'}
G {'a': 'H', 'b': 'I'}
H {'a': 'D', 'b': 'E'}
I {'a': 'B', 'b': 'C'}


Minimized DFA accepting states: {'F', 'H', 'I', 'G'}
```



确定性有限自动机（DFA）最小化

整个过程我们通过将状态分为等价类，然后创建新的状态标识符和状态映射，以实现 DFA 的最小化。这有助于简化 DFA 并减少其状态数，同时保持其等价性。最小化后的 DFA 能够执行相同的语言，但其状态数更少。

划分等价类

```
    # 步骤1: 划分等价状态类
    def split_into_equivalence_classes(states, transitions,
alphabet):
        # 初始化等价状态类，包括接受状态和非接受状态
        equivalence_classes = [accepting_states, set(states) -
accepting_states]
        new_equivalence_classes = []

        # 使用迭代循环直到不再有新的分解发生
        while equivalence_classes != new_equivalence_classes:
            new_equivalence_classes = equivalence_classes.copy()
            for i in range(len(equivalence_classes)):
                for symbol in alphabet:
                    group1 = set()
                    group2 = set()

                    # 检查每个状态的目标状态是否在同一等价类中
                    for state in equivalence_classes[i]:
                        target = transitions[state][symbol]
                        if target in equivalence_classes[i]:
                            group1.add(state)
                        else:
                            group2.add(state)

                    # 如果分成两组，则更新等价状态类
                    if group1 and group2:
                        new_equivalence_classes[i] = group1
                        new_equivalence_classes.append(group2)

            equivalence_classes = new_equivalence_classes

        return equivalence_classes
```

1. 初始化等价状态类（`equivalence_classes`）：首先，我们将初始的等价状态类初始化为两个集合，其中一个包含接受状态（`accepting_states`），另一个包含非接受状态。这是初始的等价状态划分。
2. 迭代循环直到不再有新的分解发生：使用一个 `while` 循环，我们不断迭代执行以下步骤，直到不再有新的分解（即 `equivalence_classes` 不再变化）。
3. 对每个等价类进行迭代：对每个等价类执行以下操作，通过循环遍历每个输入符号（字母表中的符号）：

   a. 创建两个空集合 `group1` 和 `group2`，用于存储当前等价类中的状态。

   b. 检查每个状态的目标状态是否在同一等价类中：对于当前等价类中的每个状态，查找它通过当前输入符号转移到的目标状态。如果目标状态仍在当前等价类中，则将该状态添加到 `group1` 中，否则添加到 `group2` 中。

   c. 如果分成两组（`group1` 和 `group2` 都非空），则将当前等价类更新为 `group1`，并将 `group2` 添加到等价状态类列表中。这样，等价状态类进行了分解。
4. 循环迭代：直到不再发生新的分解，即 `equivalence_classes` 不再变化，算法结束。此时，`equivalence_classes` 中包含的就是最小化后的等价状态类。
5. 返回最小化后的等价状态类列表：函数返回包含最小化后的等价状态类的列表。这些等价状态类将用于下一步骤，即获取新状态映射。

获取新状态映射

```python
# 步骤2: 获取新状态映射
def get_new_state_mapping(equivalence_classes):
    state_mapping = {}
    for i, eq_class in enumerate(equivalence_classes):
        # 创建新状态标识符，例如 'S0', 'S1', 等
        new_state = f'S{i}'
        for state in eq_class:
            # 映射每个等价状态到新状态标识符
            state_mapping[state] = new_state
    return state_mapping
```

1. 初始化一个空字典 `state_mapping` 用于存储新状态和旧状态之间的映射关系。
2. 对于每个等价状态类（`eq_class`）执行以下操作： a. 使用 `enumerate` 函数迭代等价状态类列表，获取索引 `i` 和等价状态集合 `eq_class`。

b. 创建新状态标识符 `new_state`，这些标识符通常以 'S' 开头，后面跟着索引号 `i`，例如 'S0'、'S1' 等。

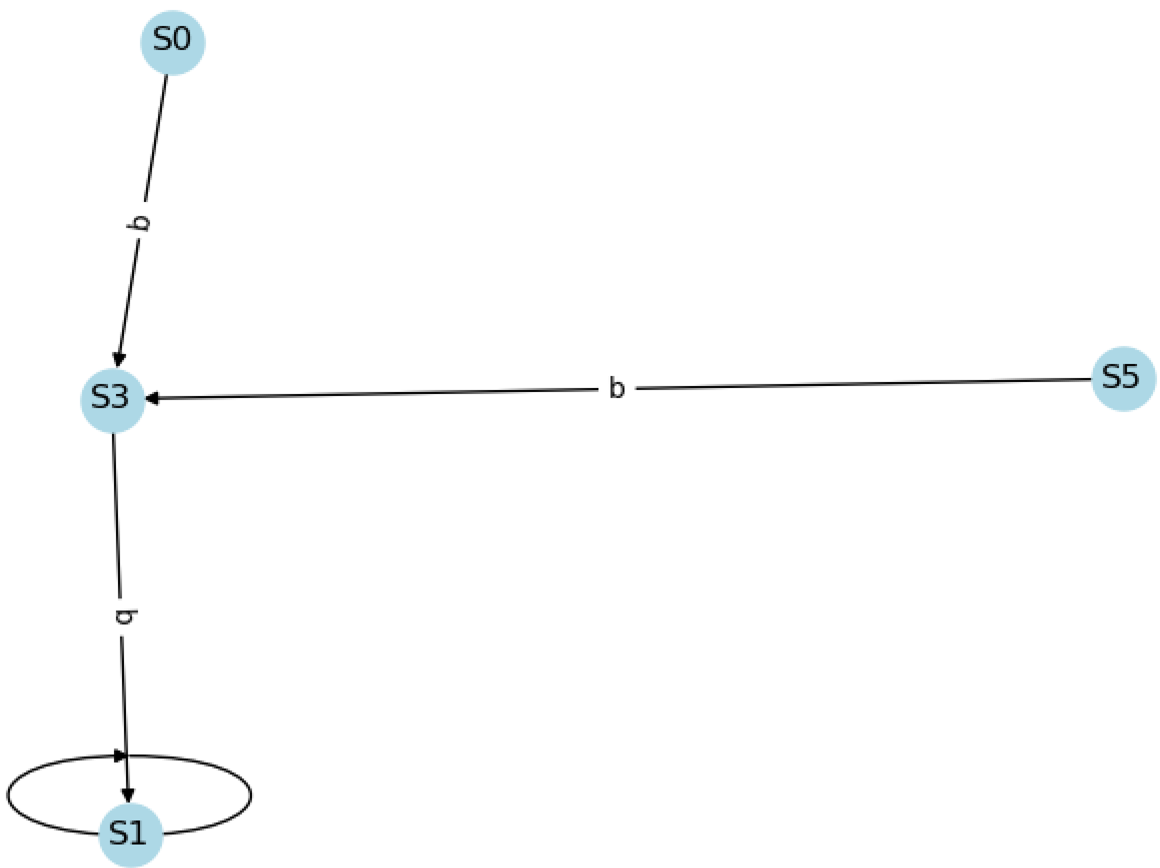
c. 遍历等价状态类中的每个旧状态（`state`）。

d. 将旧状态（`state`）映射到新状态标识符（`new_state`），将这个映射关系存储在 `state_mapping` 字典中。

3. 返回 `state_mapping` 字典，其中包含了旧状态和新状态之间的映射。这将用于下一步骤，即构建新的最小化 DFA 转移函数。

实验结果

```
Minimized DFA transitions: {'S1': {'a': 'S1', 'b': 'S1'}, 'S5':
{'a': 'S3', 'b': 'S3'}, 'S0': {'a': 'S3', 'b': 'S3'}, 'S3': {'a':
'S1', 'b': 'S1'}}
Minimized DFA start state: S1
Minimized DFA accepting states: {'S3', 'S0'}
```



实验总结

已经实现

1. 通过输入NFA的节点和转移函数，可以将接受集找出并将NFA转换成DFA，并绘制DFA有向图。
2. 输入DFA的节点和转移函数，可以将改DFA最小化并输出相关信息以及最小DFA图。

需要改进

1. 初始的接受集需要程序员给出，程序无法自己识别。
2. 绘制图像时，无法标记出接受集，同时图中的环无法显示标记。