形式语言与自动机 第一次实验 NFA的带路径执行 实验文档

实验概述

本实验需要大家以编程的方式完成，目标是编写一个NFA的执行器，能够在给定的NFA上执行指定的输入字符串。若输入字符串被NFA接受，则还需要额外返回NFA接受它的一条路径。

编程语言相关要求

本实验需要大家在**C++**或**Python**语言中任选一种进行完成。
对于每种语言，我们都提供了一套SDK（半完成的代码框架），大家只需按要求完成相应的函数即可。

关于外部依赖，在第一次实验中，**不允许大家使用任何的外部依赖**。
即C++语言只能使用标准库(stdlib)，Python语言只能使用系统库，不可以引入任何第三方库（无论是以源代码复制、pip、cmake或是任何其他形式都不可以）。此外，标准库或系统库中与正则表达式有关的库（如C++的std::regex，Python的re）也不可以使用。

实验框架内容概述

你收到的实验框架中包含的内容和含义描述如下表：

目录名	描述
cpp	C++语言的编程框架。具体的用法参见实验具体说明部分。
python	Python语言的编程框架。具体的用法参见实验具体说明部分。
cases	存放测试样例的文件夹。每个测试样例是一个txt文件。
judge	内含不同平台上的评测器的可执行文件。命名均为judge-操作系统-架构，请自行选择合适的评测器运行。如果没有你合适的平台的可执行文件，请直接联系助教。

作业形式、评分规则和提交方法

第一次实验采取全自动评测的形式。
自动评测的含义是：你的程序将被多次调用、输入多个不同的测试样例进行测试。对每个测试样例，若你的程序在规定时限内正常返回并给出正确的结果，则该测例得分，否则不得分。
本次实验**不需要提交文档**。

提交方法

请在网络学堂的DDL前，将你的作业提交到**网络学堂**。提交的内容为一个压缩包（建议最好是`.zip`格式），内容包含：

- 若你是使用C++语言完成，请提交：
  - `cpp`文件夹
    - 请确保删除了`cpp`文件夹内的中间产物文件夹如`build`、`cmake-build-*`等。
    - 请勿删除`CMakeLists.txt`，否则你的代码将无法编译！

    - 在你编程的平台上编译出的可执行文件。Windows请命名为`nfa.exe`，Mac和Linux请命名为`nfa`（即CMake的默认编译输出名）
        - 此项用于在助教无法亲自编译你的代码的情况下作为参考。如果助教能够成功编译你的代码，则不会使用你提交的可执行文件，此种情况下即使你未提交可执行文件也不会扣分。
    - 一个简短的TXT文件，命名为`README.txt`，说明你的编程平台（如操作系统、架构、编译器等）。如果有其他想给助教留言的也可以写在这里。
    - 请不要提交除上述内容外的其他东西。特别是注意不要提交`python`文件夹，否则可能造成评测器混淆你的编程语言，严重影响评测分数。
- 若你是使用Python语言完成，请提交：
    - `python`文件夹。**确保`nfa.py`文件装在python文件夹里，不要仅提交一个`nfa.py`文件**，否则评测器可能会找不到！
    - 一个简短的TXT文件，命名为`README.txt`，说明你的编程平台（如操作系统、架构、Python版本等）。如果有其他想给助教留言的也可以写在这里。
    - 请不要提交除上述内容外的其他东西。特别是注意不要提交`cpp`文件夹，否则可能造成评测器混淆你的编程语言，严重影响评测分数。

评分规则

本实验的评分规则为：

- 自动评测 100%
    - 公开测例 60% (20个，每个测例分值均等)
        - 已包含在本次下发的实验框架中，同时提供了自我测评用的评测器。
        - **使用评测器得到的自测得分仅供参考**，不作为得分的直接依据。
    - 隐藏测例 40% (每个测例分值均等)
        - 不会公开给同学。将在DDL后由助教进行测试。
    - 所有的测例，无论是公开还是隐藏，均将**由助教在DDL后统一运行你的代码进行测试并计算得分**。得分以助教评测的结果为准。
- 减分项：如你存在下列问题，可能会被额外进行惩罚性的减分。
    - 抄袭：**本实验和之后的所有实验均严禁抄袭**。抄袭者最严重将被处以所有实验全部0分的惩罚。不能给出合理解释的代码高度雷同也被视为抄袭。
    - 攻击评测机：禁止用任何方式攻击评测机，包括但不限于尝试访问、修改与自己的实验无关的文件、执行恶意代码、尝试提权等行为。违反者视情节，最严重将被处以所有实验全部0分的惩罚，如涉及违纪违法还将上报学校等有关部门处理。
    - 使用非正常手段通过测例：包括但不限于针对特定的输入直接匹配输出，通过联网、调用评测机等手段从外部来源获取答案等。违反者将被扣除所有以非正常手段通过的测例的得分。
    - 未按要求提交：请严格按照上面的提交要求提交作业。
        - 若未按要求提交引起评测器无法自动评测分数的（如同时提交cpp和python文件夹等行为），将被至多扣除本次实验总分的20%。
        - 其他情况，至多扣除本次实验总分的10%。

迟交政策

- 每迟交一天，分数扣减5%，至多扣减50%。
- 扣减是在正常方法计算的应得分的基础上按比例扣减的。
    - 例如，迟交3天，程序经过评测应得90分，则实际最终得分为90*(1-5%*3)=76.5分（分数舍入到小数点后一位）。

其他

- 保证无论是自动机的状态转移规则的字符，还是输入的字符串中，都不会包含NULL字符`\0`和换行符`\r` `\n`。
- 程序“正常返回”的定义：你的程序正常结束执行，进程返回码为0。
- 助教评分时使用的评测环境：
    - Ubuntu 22.04 LTS (in docker)
    - Intel i7-12700K (5.0GHz)
    - Python 3.11
- 关于程序限时：本实验给出的限时可以说是非常宽松的。
    - 对于C++语言，每个测例限时5s；对于Python语言，每个测例限时10s。
    - 限时一律以程序在助教的评测机上的用时为准，均包括IO时间。
    - 请注意，提供给你的自测用评测器**只实现了统计限时，但并未实现超时杀死进程的机制**。
        - 即，无论你的程序运行多久，评测器都会一直等待直到你的进程退出，然后计算你的程序执行的用时是否超时。
        - 超时的测例在评测器中会分类在“发生异常”类别中。

实验具体说明

关于转移规则和Rule对象

在自动机输入文件中，转移规则可由类似`0->1 a b \d`的形式表达，表示状态0可通过字符`a`、字符`b`或特殊字符`\d`转移到状态1。
但是无论在C++还是Python的SDK中，一个`Rule`对象只能表示经由一种字符的转移。也就是说，上面这一行在`NFA`对象的`rules`中，实际上会被拆成三条`Rule`，分别对应`a` `b` `\d`三个字符。
本实验中，要求大家支持以下四类转移规则：

- 普通转移。转移字符是**单个**ASCII字符。
    - 例如`0->1 a`，将被对应为`rules[0]`中的一个`Rule`对象`dst=1, type=NORMAL, by="a"`，仅匹配字母`a`。
- 字符区间转移。转移字符是ASCII字符的区间，如A-Z。
    - 例如`1->2 A-Z`，将被对应为`rules[1]`中的一个`Rule`对象`dst=2, type=RANGE, by="A", to="Z"`，匹配任意大写字母。
- 特殊字符转移。转移字符为一些特殊字符，需要支持的所有特殊字符详见下表。
    - 例如`2->3 \d`，将被对应为`rules[2]`中的一个`Rule`对象`dst=3, type=SPECIAL, by="d"`（注意`by`中没有`\`，只有`d`），匹配任意数字。

字符	等价于*	说明
`\.`	`.`	匹配除换行符`\r` `\n`以外的任意单个字符。

字符	等价于*	说明
`\d`	`[0-9]`	匹配任何数字。
`\s`	`[ \f\n\r\t\v]`	匹配任何空白字符，具体包括哪些字符请参考其等价形式。
`\w`	`[A-Za-z0-9_]`	匹配字母、数字、下划线。
`\D`	`[^0-9]`	匹配`\d`不匹配的任何字符。
`\S`	`[^ \f\n\r\t\v]`	匹配`\s`不匹配的任何字符。
`\W`	`[^A-Za-z0-9_]`	匹配`\w`不匹配的任何字符。

    - * 指等价于标准正则表达式中的什么表达式
- epsilon-转移。
    - 例如`3->4 \e`，将被对应为`rules[3]`中的一个`Rule`对象`dst=1, type=EPSILON`，是一个epsilon-转移。

代码框架公共说明——对所有的语言都适用

- cases中的测试样例均为文本文件，内含NFA的定义和输入字符串。这些内容是人类可读的（PPT里有更具体的含义介绍），如有需要，你也可以在其基础上进行修改/编写自己的测例进行测试。
    - 如你需要自行修改/构造测试样例，**建议复制出来改动而不是在测例上直接修改**，否则如果你忘改回去了可能会造成自测分数与最终分数不一致
- 你所需要做的事是完成NFA类的exec函数。在代码中已经使用TODO注释为你标记好。
    - 你的程序并不需要亲自从stdin中读取输入，也不需要亲自向stdout中写入结果。
    - 框架已经实现好了读取并解析文本输入，构造NFA类的对象，再调用exec方法，并将exec方法返回的结果输出到stdout的逻辑。
        - 如你感兴趣，可看`main-nfa.cpp`或`nfa.py`文件最下面的`if __name__ == '__main__'`部分。
    - 你的程序**尽量不要在stdout中打印输出**，如果确实需要打印，请**尽量打印在stderr中**。
        - 至少，请确保你打印的内容结尾有换行符。评测器的逻辑是读取stdout的最后一行作为评测的依据。
- 程序的调用方法分为两种：
    - 若不传入任何参数，则程序将从stdin中读取输入。（也是评测器评测时所用的方式）
    - 或者，你可以传入一个参数，是输入文件的路径。此时程序将改为从你指定的文件路径中读取输入。（当你想要具体的在某个测例上执行和调试时，这种方法更为方便些）
- **仔细阅读框架代码的注释！** 很多问题，包括类的含义、函数的含义、返回值的方式等，都可以在框架代码的注释中可以找到答案。
    - 框架中已经定义好了一些和函数，类内也已经定义好了一些成员变量和方法。不建议大家修改这些已经定义好的东西。
    - 但是，你可以自由地增加新的函数、类等，包括可以在已经定义好的类自由地添加新的成员变量和方法。如果你确实需要，也同样可以增加新的文件（但C++语言请注意将新增的文件加到`CMakeLists.txt`的NFA target里）。

C++语言

编译执行方法

本框架的C++语言部分使用CMake作为构建的工具。

IDE使用提示

诸如CLion、Visual Studio等IDE均支持CMake。一般来说，你只需打开项目，就能够顺利的完成编译、运行和调试。 不同的IDE，加载项目、编译和运行程序以及修改程序运行配置（为程序传参）的方法略有不同，以下仅就助教了解的一些IDE的用法进行提示：

- CLion：
  - 使用起来比较简单，直接打开cpp文件夹，就会自动配置CMake项目，出现名称类似于`nfa | Debug`的运行目标，点击运行即可。
  - 配置命令行参数：运行目标处的小三角——Edit Configurations——弹出的窗口中修改`Program arguments`即可。参数示例：`../../cases/01.txt`
- Visual Studio：
  - 应该也是打开文件夹就会自动配置CMake项目，然后可以找到名为`nfa.exe`的启动项，点击运行即可。
  - 配置命令行参数：相对麻烦一些。请参考这个文章。以下是助教经过测试可用的launch.json的示例：

```
{
  "version": "0.2.1",
  "defaults": {},
  "configurations": [
    {
      "type": "default",
      "project": "CMakeLists.txt",
      "projectTarget": "nfa.exe",
      "name": "nfa.exe",
      "args": ["../../../../cases/01.txt"]
    }
  ]
}
```

- VSCode：
  - 需要确保安装`C/C++ Extension Pack`和`CMake Tools`两个插件。
  - 首先需要配置CMake和工具链。打开命令窗口(`Ctrl+Shift+P`)，搜索并执行命令`CMake: Configure`，然后选择合适的工具链、选择正确的(`cpp`文件夹下的)`CMakeList.txt`。然后会开始CMake配置，配置好之后，下方蓝色的状态栏会出现`CMake: [Debug]: Ready`和你的工具链的名字(不同的工具链名字差异可能很大)。如果确认状态栏出现了这两个东西，则说明CMake配置正确。
  - 然后再次打开命令窗口，搜索并执行命令`CMake: Debug`，即可开始调试。（第二次起可使用该命令的快捷键`Ctrl+F5`）
  - 配置命令行参数：请修改`.vscode`文件夹下的`settings.json`，加入以下内容：
    - 若没有此文件，直接新建即可。

    - `"externalConsole": true`的意思是使用外部的console窗口。某些工具链/debugger若不加此选项就无法看到控制台输出。但也有些工具链/debugger并不需要这个配置。可参阅这篇文章

```
{
   // ... 如果文件此前已存在并有其他的字段，请不要删除这些原来已存在的字段，直接添加下面的cmake.debugConfig字段即可；如果文件是你新建的，则删除此行注释，剩下的内容直接粘贴到文件即可
    "cmake.debugConfig": {
        "externalConsole": true,
        "args": [
            "../cases/01.txt"
        ]
    }
}
```

直接在命令行中编译运行

或者，若你想直接使用命令行进行编译，方法如下：

```
cd cpp
mkdir build # 作为编译结果（可执行文件）和各类编译中间产物存储的文件夹
cd build
cmake .. # 意思是去找上级目录（此时你在build中，上级目录就是cpp）中的CMakeLists.txt文件，据此在当前目录(build)中进行中间产物的生成。这步cmake会帮你生成好一个Makefile。
cmake --build . # 执行编译
```

执行文件的方法：（注意windows平台上是nfa.exe）

```
./nfa # 程序会从stdin中读取数据，请自行使用输入重定向 < 、管道 | 等手段为它提供输入
./nfa ../../cases/01.txt # 程序会从指定的路径读取输入。此处假定你在cpp/build文件夹下，故测试样例的相对路径应如同这个样子
```

当然，由于本次实验没有用到外部依赖，`CMakeLists.txt`也十分简单，所以你绕过CMake，直接将文件夹中的所有`.h`和`.cpp`文件作为输入进行编译也是可以的。但请注意不要改动或删除`CMakeLists.txt`，否则助教这边将无法编译你的程序。

代码结构具体描述

- `NFA`类：`nfa.h` `nfa.cpp`
    - 包括`NFA`类的定义和类中成员用到的一些结构体的定义。
    - 最重要的是`num_states` `is_final` `rules`成员变量，和`Rule` `Path`结构体。
    - 你需要实现的是`NFA::exec`函数，其参数和返回值含义均在注释上。请在`nfa.cpp`中完成其实现。
    - 你应该不需要去管`NFA::from_text`,`ostream &operator<<(ostream &os, Path &path)`等函数。这些函数是由框架自动调用的，你不需要理解其含义和查看其代码。
- 入口点文件：`main-nfa.cpp`
    - 你应该不需要去管这个文件。这个仅包含`main`函数的实现，其中会构造`NFA`类的对象和调用`exec`方法。

Python语言

运行方法

需要的Python版本应>=3.8，否则可能会无法运行。
`python`文件夹中只有一个文件`nfa.py`，就是程序的入口点。

```
python nfa.py # 程序会从stdin中读取数据，请自行使用输入重定向 < 、管道 | 等手段为它提供输入
python nfa.py ../cases/01.txt # 程序会从指定的路径读取输入。此处假定你在python文件夹下，故测试样例的相对路径应如同这个样子
# 或者，如果你使用类Unix系统(Linux、Mac)，由于nfa文件中包含了Shebang，也可以不必输入python、直接执行nfa.py
./nfa.py
```

代码具体描述：`nfa.py`

- 包括`NFA`类的定义和`NFA`中用到的`Path`、`Rule`等其他类的定义，也包含程序的入口点`__main__`代码。
- 最重要的是`NFA`类中的`num_states` `is_final` `rules`成员变量，及`Rule` `Path`类。
- 你需要实现的是`NFA`类中的`exec`函数，其参数和返回值含义均在注释上。
- 你应该不需要去管`NFA`的`from_text`，`Path`的`__str__`等函数。这些函数是由框架自动调用的，你不需要理解其含义和查看其代码。

评测器的使用方法

- 评测器位于`judge`文件夹中，是编译好的可执行程序，针对多种目标平台。请根据自己的平台选择对应的程序。
  - 文件名为`judge-xxx-yyy`，其中xxx表示操作系统，yyy表示CPU架构。Windows平台上后面还会加上表示可执行文件的扩展名`.exe`。
  - 现在常见的Windows和Linux电脑大多都是amd64架构的（无论CPU是Intel还是AMD，Intel的64位CPU架构名也叫amd64）。
  - MacOS的操作系统代号叫darwin。最新的M1和M2芯片是arm64的，而早期的Mac都是amd64的。
  - 如果你使用的平台无法在当前的`judge`文件夹中找到对应，请直接与助教联系，助教会为你补充提供对应的程序。
- 评测器的运行方法：`judge [-i input_files_or_dir1,...] [-t timeout] [--full_score score] your_process_and args...`
  - 即，需要将**调用你的程序的方式**作为参数传给评测器
    - C++语言例如：先`cd`到`judge`目录，然后

```
./judge-xxx-yyy ../cpp/build/nfa # ../cpp/build/nfa处应为你的C++代码编译出的可执行文件的路径
```

    - Python语言例如：先`cd`到`judge`目录，然后

```
./judge-xxx-yyy python ../python/nfa.py # ./judge-xxx-yyy后的部分即为你平时调用你自己的python脚本的方式。注意此处假定你在judge目录下，如在其他目录下，则python脚本的路径会有相应变化。
```

    - 此外，你还可以可选地传入命令行参数。参数的位置需要在`./judge-xxx-yyy`和调用程序的方式参数之间，多个参数时顺序没有要求。
        - `-i`：可传递输入文件，或大量输入文件所在的目录。需要传多个路径时，路径之间用逗号`,`分隔，注意逗号前后不要有空格。不传此参数时会默认使用`cases`文件夹下的所有文件。
        - `-t`：可传递每个测例的限制时长。默认为C++语言5s，Python语言10s。
        - `--full-score`: 可传入评测器最后显示分数的满分。默认为60。
    - 评测器会对每个输入样例分别调用一次你的程序，并把评测的结果实时打印出来。全部评测完成后，还会给出统计和最终得分。
- Mac用户请注意：你在首次运行评测器时，可能会遇到`operation not permitted`的错误。
    - 这是因为macOS的 Gatekeeper 机制阻止了未知开发者的可执行文件运行。为此，你需要在`judge`文件夹运行下列命令解除 Gatekeeper 限制：

```
sudo xattr -r -d com.apple.quarantine ./*
```