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

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，在全部的实验中，**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，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。

代码框架内容概述

代码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的C++语言部分。
python	Python语言的编程框架。具体的用法参见实验具体说明的Python语言部分。
cases	存放测试样例的文件夹。每个测试样例是一个txt文件。

提交方法

请提交到本课程的OJ上（须在清华校园网环境下访问，校外请用SSLVPN客户端）。登录OJ所用的账号密码已经通过网络学堂下发给大家。

请以 **.zip格式**的压缩包进行提交，压缩包中的内容请遵守以下规则：

- 若你是使用C++语言完成，请仅提交`src`文件夹下的内容。
  - 请确保删除了编译产物文件夹如`build`、`cmake-build-*`等。
  - 请勿删除`CMakeLists.txt`，否则你的代码将无法编译！
- 若你是使用Python语言完成，请提交`python`文件夹下的内容。
  - > 尽管第一次实验中python文件夹下只有nfa.py一个文件，但由于下次实验我们将会引入更多的文件，为了使提交格式统一，本次实验也请你提交压缩包而不是单个文件。
- 不需要提交文档。

提交时请注意，不要使用OJ中的“在线编程模式”，而是**点击“递交”**，然后通过 **“Or upload a file”按钮**上传你的压缩包，如下图所示。

评分规则

本次实验占据实验部分总成绩的25%。

本次实验的评分规则如下：

- 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。在DDL前你提交到OJ时，OJ显示的分数即为本部分的分数。
  - 最终的成绩以DDL后使用全部公开和隐藏测例重测得到的成绩为准。
- 隐藏测例 40% (每个测例分值均等)
  - 不会公开给同学。将在DDL一段时间后，与全部公开测例一起进行重测，重测后显示的得分即为最终总得分。
- 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：**本实验和其他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。
  - 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。

迟交政策如下：

- 每迟交一天，分数扣减5%，至多扣减60%。
  - 接受补交的最晚日期另行通知。
- 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，正常计算的应得分为90分，则实际最终得分为90*(1-5%*3)=76.5分。
- 如你迟交较多、OJ上的作业窗口已经关闭，则你可以先通过“题库”进行公开测例的自助测试。然后在你准备好后，单独联系助教进行最终的十成测。
  - 在此种情况下，你只有一次十成测的机会。

其他

- 第一次实验，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，都只包含ASCII字符（字节值0~127），且不会包含NULL字符`\0`和换行符`\r` `\n`。

- 数据约定：对全部的测例，NFA状态数<=200，总转移规则条数<=1000，输入字符串长度<=10000。
  - 事实上，你只需要用普通的思路，不太需要什么特殊的优化技巧，就足以完成
- 时空限制：时间：C++ 2s，Python 4s；内存256MiB。
- OJ评测机评测环境：
  - Ubuntu 22.04.3 LTS (in docker)
  - Intel i7-12700K (5.0GHz)
  - Python 3.11.8
  - GCC 13.2.0

实验具体说明

任务说明（必读）

- **本次实验中，你只需要完成一个函数**（在代码中已经使用**TODO注释**为你标记好）：
  - `NFA`类的`exec`函数。
- 你只需要完成上述的函数即可，不需要自己处理标准输入输出相关的问题。
  - 你的代码**不应在stdout中打印任何额外的输出**，如果确实需要打印，请**打印在stderr中**。
- 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，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。
- 运行程序时，请将**单个测试样例文件**作为**唯一的参数**传入。
  - 具体的方法见对应语言的框架说明：C++框架说明 Python框架说明
  - 或者，若不传入任何参数，则程序将从stdin中读取输入。如果你懂得如何把测试样例通过stdin输入进去（如重定向`<`、`cat |`等），也可采取此方法。
- **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`add_executable(nfa ...)`里）。
- 提示：在构造`Path`对象时，必须**保证`consumes`的长度必须为`states`的长度-1**！
  - `consumes[i]`表示`states[i]`迁移到`states[i+1]`时所消耗的字母。
  - 若从`states[i]`迁移到`states[i+1]`是通过不消耗字母的ε-转移，则`consumes[i]`**应设为空串**！

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。
`\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
- ε-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ε-转移。

使用exportJFLAP函数导出NFA

为了方便大家的调试，代码框架中提供了**将NFA导出成为JFLAP**的.jff格式文件的工具函数`exporJFLAP`，导出的文件可以用JFLAP打开并可视化。
利用该工具函数可以实现可视化地查看你所生成的NFA，希望该工具函数能够帮到大家的调试。

注意： 该函数仅用于大家本地进行测试使用。**提交OJ的版本中切勿包含此函数的调用**，否则会造成OJ无法通过测试点（如PE 输出格式错误）！

- C++: 位于`util.h`中，`void exportJFLAP(const NFA &nfa, std::string filename, int distance, int yLevels)`
- Python: 位于`nfa.py`中，`def exportJFLAP(nfa: NFA, filename, distance, yLevels)`
- 其中`nfa`参数必需，是要导出的NFA类的对象；其他参数可选，具体用法详见注释。

默认情况下该函数未被调用，如需使用该函数，只需在合适的地方（如NFA::exec的开头），加上对该函数的调用，形如：`exportJFLAP(nfa)`即可。

C++语言框架说明

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：运行目标处的小向下箭头——编辑配置(Edit Configurations)——弹出的窗口中修改`程序实参(Program arguments)`即可。
    - 参数示例：若`工作目录(Working Directory)`在`cpp`目录，则`程序实参`对应应为`../cases/01.in`；类似的，若`工作目录`在实验代码包的根目录（即比`cpp`再上一层），则`程序实参`对应应为`cases/01.in`。
    - 如果你不能理解上面所说的配置的原因，请自行学习相对路径的概念。如果你还是搞不明白，则请直接在`程序实参`中**使用绝对路径**。
- Visual Studio：
  - 通常情况下，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：（以VS2022为例。其他版本可能按钮名字略有区别，但大同小异）
    - 在右侧的"解决方案资源管理器"中，找到`src`目录下的`CMakeLists.txt`，并右击这个文件。
    - 右击弹出的菜单中，点击"添加调试配置"，然后在弹出窗口中选择"默认"类型。
    - 此时会创建一个名为`launch.vs.json`的文件（位于`.vs`目录下），并在左侧自动打开。参照下方的示例配置。
    - 后面需要再次打开`launch.vs.json`时，右键`CMakeLists.txt`——点击"打开调试和启动设置"。
  - 关于`type`和`project`字段: 请**保留这两个字段**，并维持VS自动创建出它们时的取值，**不要删除**这两个字段。
  - 关于`args`的取值：VS默认的工作路径是编译出的可执行文件的所在路径，即`cpp\src\out\build\x64-Debug\nfa.exe`。因此根据相对路径的规则，应该向上跳5级才能到实验代码包根目录。
    - 如果由于你的版本/环境特殊等原因，使用这个相对路径无法找到测例文件，则请直接**使用绝对路径**。

```
{
  "version": "0.2.1",
  "defaults": {},
  "configurations": [
    {
      // 重要！请保留原本的"type"和"project"字段，不要删除！且复制过去后请移除此行注释，否则配置不会被识别！
      "projectTarget": "nfa.exe",
      "name": "nfa.exe",
      "args": ["../../../../../cases/01.in"]
    }
  ]
}
```

- 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（老版VSCode：`CMake: [Debug]: Ready`和你的工具链的名字；新版VSCode：`build`按钮，debug虫子图标和运行图标）。如果确认状态栏出现了上述内容，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`（新版VSCode可以点击下方蓝色状态栏上的debug虫子按钮），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。反之，如果此文件已存在并有其他的字段，则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。
    - **关于`args`的相对路径(重要)**：由于下面的配置中使用了`"cwd": "${workspaceRoot}"`，限定了运行时的工作目录为项目根目录（即你打开项目时所选的的文件夹），因此`args`参数的取值视乎你项目根目录的位置会有所不同。具体而言：
      - 若你的项目根目录在实验代码包的根目录（即比`cpp`再上一层），则测例的相对路径此时为`cases/01.in`。
      - 若你项目根目录在`cpp`目录，则测例的相对路径此时为`../cases/01.in`。
      - 如果你不能理解上面所说的配置的原因，请自行学习相对路径的概念。如果你还是搞不明白，则请直接在`程序实参`中**使用绝对路径**。
  - 小提示：对MSVC等部分工具链，使用调试时的控制台输出会出现在“DEBUG CONSOLE”中。如果你还是找不到的话，可以参阅这篇文章的方法，加入`"externalConsole": true`。

```
{
    // ... 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
    "cmake.debugConfig": {
        "cwd": "${workspaceRoot}",
        "args": ["../cases/01.in"] // 取值视乎你项目根目录的位置会有所不同，详见上方的文
字说明
    }
}
```

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 ../src # 意思是去找../src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 . --target nfa # 执行编译
```

执行文件的方法：（注意windows平台上是nfa.exe）

```
./nfa ../../cases/01.in # 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
# 或者，也可以不带参数执行
./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
```

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语言框架说明

运行方法

建议Python版本>=3.8。
`python`文件夹中只有一个文件`nfa.py`，就是程序的入口点。

```
python nfa.py ../cases/01.in # 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，也可以不带参数执行
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
```

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。