

软件工程实验报告

实验四：等价判断

院系：人工智能学院

姓名：方盛俊

学号：201300035

班级：人工智能 20 级 2 班

邮箱：201300035@smail.nju.edu.cn

时间：2022 年 11 月 4 日

目录

- 实验四：等价判断
 - 目录
 - 一、版本控制
 - 1. Git Init
 - 2. 编写 `.gitignore` 文件
 - 3. Git Add
 - 4. Git Commit
 - 5. Git Diff
 - 6. Git Reset
 - 7. Git Revert
 - 8. Git Stash
 - 8. Git Checkout
 - 9. Git Merge
 - 10. Git Rebase
 - 11. Git Cherry Pick
 - 12. GitHub 远程仓库
 - 13. 分支合并图
 - 二、代码架构
 - 1. 执行比较 (diff)
 - 2. 生成样例 (generator)
 - 3. 中间表示与等价类 (cluster)
 - 4. 输入 (input)
 - 5. 并行多进程计算 (paracomp)
 - 6. 输出 (output)
 - 三、运行流程
 - 四、优秀设计
 - 1. 执行模块的扩展性
 - 2. 基于并查集的等价类
 - 3. 测试样例的多样性
 - 4. 并行多进程计算
 - 5. 保存中间状态
 - 6. 为后续的前端界面预留了接口

一、版本控制

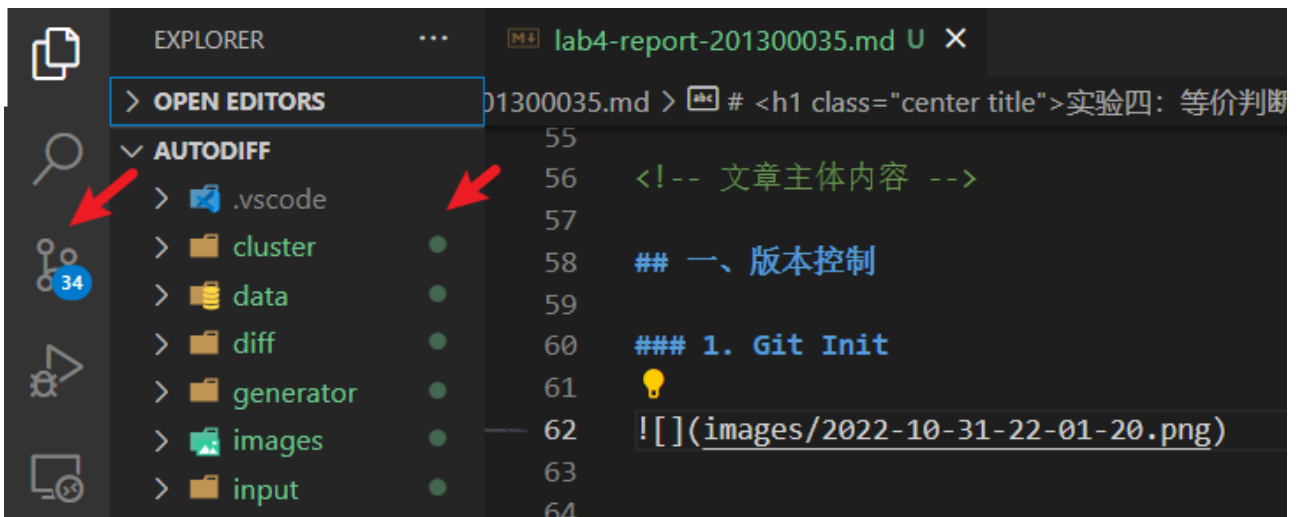
在进行版本控制之前，先按照上一次实验书写的实验报告，初始化好项目结构：

```
.
├── .gitignore (不加入 Git 里的文件)
├── lab4-report-201300035.md (实验报告)
├── main.py (项目主入口)
├── requirements.txt (Python 依赖文件)
├── cluster
├── data (示例数据)
│   ├── input
│   │   ├── 4A
│   │   │   ├── 101036360.cpp
│   │   │   ├── 117364748.cpp
│   │   │   ├── 127473352.cpp
│   │   │   ├── 134841308.cpp
│   │   │   ├── 173077807.cpp
│   │   │   ├── 48762087.cpp
│   │   │   ├── 84822638.cpp
│   │   │   ├── 84822639.cpp
│   │   │   └── stdin_format.txt
│   │   └── 50A
│   │       ├── 138805414.cpp
│   │       ├── 142890373.cpp
│   │       ├── 164831265.cpp
│   │       ├── 21508887.cpp
│   │       ├── 21508898.cpp
│   │       ├── 21715601.cpp
│   │       ├── 29019948.cpp
│   │       ├── 30534178.cpp
│   │       ├── 31034693.cpp
│   │       ├── 33794240.cpp
│   │       ├── 36641065.cpp
│   │       ├── 45851050.cpp
│   │       └── stdin_format.txt
│   └── output
├── diff
├── generator
├── images
├── input
├── output
├── paracomp
└── server
```

1. Git Init

执行 `git init` 初始化项目的 Git 仓库。

```
(base) PS C:\Users\OrangeX4> cd C:\Project\autodiff
(base) PS C:\Project\autodiff> git init
Initialized empty Git repository in C:/Project/autodiff/.git/
```



可见 VS Code 版本控制面板和目录树标绿，说明已经初始化完毕，正在等待将文件加入 Git 仓库中。

2. 编写 `.gitignore` 文件

编写 `.gitignore` 文件，将无需加入 Git 仓库的文件标在这里。

```
# Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# Executables
*.exe
*.out

# IDE/Editor configuration
.vscode/

# Filesystem file
.DS_Store
```

这是很重要的一步，可以避免无用文件被加入 Git 仓库里 (例如一些需要实时生成的二进制文件和可执行文件，以及与 IDE 紧耦合的配置文件)。

3. Git Add

执行 `git add *` 将所有文件加入到 Git 仓库中。

```
(base) PS C:\Project\autodiff> git add *
```

执行前的 `git status` :

```
(base) PS C:\Project\autodiff> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        cluster/
        data/
        diff/
        generator/
```

执行后的 `git status` :

```
(base) PS C:\Project\autodiff> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   cluster/__init__.py
        new file:   data/input/4A/101036360.cpp
        new file:   data/input/4A/117364748.cpp
        new file:   data/input/4A/127473352.cpp
```

4. Git Commit

执行 `git commit -m "feat: init"` 进行一次初始化的 Commit, 并附上 Commit 信息 `feat: init`。

```
(base) PS C:\Project\autodiff> git commit -m "feat: init"
[master (root-commit) ef06984] feat: init
35 files changed, 298 insertions(+)
create mode 100644 .gitignore
create mode 100644 cluster/__init__.py
create mode 100644 data/input/4A/101036360.cpp
create mode 100644 data/input/4A/117364748.cpp
create mode 100644 data/input/4A/127473352.cpp
create mode 100644 data/input/4A/134841308.cpp
```

执行前的 `git status` :

```
(base) PS C:\Project\autodiff> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   cluster/__init__.py
        new file:   data/input/4A/101036360.cpp
        new file:   data/input/4A/117364748.cpp
        new file:   data/input/4A/127473352.cpp
```

执行后的 `git status` :

```
(base) PS C:\Project\autodiff> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   lab4-report-201300035.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        images/2022-10-31-22-25-10.png

no changes added to commit (use "git add" and/or "git commit -a")
```

5. Git Diff

`main.py` 的内容原本为:

```
print('Hello World')
```

为了进行一次 Commit, 我们将其修改为:

```
print('Hello World')
print('Hello Software Engineering')
```

执行 `git add *` 后执行 `git commit -m "feat: change main.py"`, 将变动 Commit 到 Git 仓库中。

执行 `git diff` 我们可以发现我们的更改切实地被加入到了 Git 仓库中。

```
(base) PS C:\Project\autodiff> git diff
diff --git a/lab4-report-201300035.md b/lab4-report-201300035.md
index 9284f95..28c865a 100644
--- a/lab4-report-201300035.md
+++ b/lab4-report-201300035.md
@@ -176,3 +176,5 @@ print('Hello World')
 print('Hello Software Engineering')
```
```

## 6. Git Reset

Git Reset 命令用于重置当前 HEAD 到指定的版本。

执行 `git log` 我们可以看出我们当前有 `feat: init` 和 `feat: change main.py` 这两条 Commit。

```
(base) PS C:\Project\autodiff> git log
commit 140993048c5c8d2464c4aa64b9d770c8cac45d84 (HEAD -> master)
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 22:29:34 2022 +0800

 feat: change main.py

commit 1ad1f9c5343a5ef3da9e099c090bda62970a0d1e
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 22:25:53 2022 +0800

 feat: init
```

执行 `git reset 1ad1f9` 我们就可以恢复到 `feat: init` 这一条 Commit 所在的位置。

```
(base) PS C:\Project\autodiff> git reset 1ad1f9
Unstaged changes after reset:
M lab4-report-201300035.md
M main.py
```

执行 `git log`，我们发现我们确实已经回到了只有一条 Commit 的状态。

```
(base) PS C:\Project\autodiff> git log
commit 1ad1f9c5343a5ef3da9e099c090bda62970a0d1e (HEAD -> master)
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 22:25:53 2022 +0800

 feat: init
```

## 7. Git Revert

使用 Git Revert 命令和 Git Reset 很类似，均是要恢复到之前的某些版本，但是 Git Revert 的好处在于，会把之前的 commit history 给保留下来，并把这次撤销作为一个新的 Commit。

执行 `git reset 1409930` 来恢复 `feat: change main.py` 这条 Commit, 并执行 `git log` 显示:

```
(base) PS C:\Project\autodiff> git reset 1409930
Unstaged changes after reset:
M lab4-report-201300035.md
(base) PS C:\Project\autodiff> git log
commit 140993048c5c8d2464c4aa64b9d770c8cac45d84 (HEAD -> master)
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 22:29:34 2022 +0800

 feat: change main.py

commit 1ad1f9c5343a5ef3da9e099c090bda62970a0d1e
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 22:25:53 2022 +0800

 feat: init
```

执行 `git revert HEAD`, 我们就能撤销当前版本的修改, 恢复上一个版本, 并在不改变 commit history 的情况下, 创建一个新的 Commit。

```
(base) PS C:\Project\autodiff> git revert HEAD
[master 800357d] Revert "feat: change main.py"
 4 files changed, 2 insertions(+), 29 deletions(-)
 delete mode 100644 images/2022-10-31-22-25-10.png
 delete mode 100644 images/2022-10-31-22-26-22.png
```

我们可以看出, `main.py` 文件的内容也成功回退到上一个版本了。

执行 `git log` 我们可以更清晰地看出我们做的操作:



```
(base) PS C:\Project\autodiff> git log
commit 800357d1fe7db268a45ba5681e8ffc3264330415 (HEAD -> master)
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 23:20:58 2022 +0800

 Revert "feat: change main.py"

 This reverts commit 140993048c5c8d2464c4aa64b9d770c8cac45d84.

commit 140993048c5c8d2464c4aa64b9d770c8cac45d84
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 22:29:34 2022 +0800

 feat: change main.py

commit 1ad1f9c5343a5ef3da9e099c090bda62970a0d1e
Author: OrangeX4 <318483724@qq.com>
Date: Mon Oct 31 22:25:53 2022 +0800

 feat: init
```

再次执行 `git revert HEAD`，我们可以发现 `main.py` 又恢复到了最新版本，这大概就是 `git revert` 操作的“负负得正”。

```
(base) PS C:\Project\autodiff> git revert HEAD
[master db3adbb] Revert "Revert "feat: change main.py""
4 files changed, 29 insertions(+), 2 deletions(-)
create mode 100644 images/2022-10-31-22-25-10.png
create mode 100644 images/2022-10-31-22-26-22.png
```

## 8. Git Stash

有时，当你在项目的一部分上已经工作一段时间后，所有东西都进入了混乱的状态，而这时你想要切换到另一个分支做一点别的事情。问题是，你不想仅仅因为过会儿回到这一点而为做了一半的工作创建一次提交。针对这个问题的答案是 `git stash` 命令。

这是 Git 官方文档对 Git Stash 命令的解释。事实上我也确实用到了这个命令。

在上一小节的 Git Revert 命令演示中，由于我是在同步编写实验报告，所以工作区实际上有一些变动，这导致我无法正常 `git revert`。

```
(base) PS C:\Project\autodiff> git revert HEAD
error: Your local changes to the following files would be overwritten by merge:
lab4-report-201300035.md
Please commit your changes or stash them before you merge.
Aborting
fatal: revert failed
```

这种时候，我就可以执行 `git stash`，将我对实验报告的修改暂存了起来，这样工作区就又恢复了干净的状态，同时我们可以用 `git stash list` 查看我们放在栈上的暂存修改。

```
(base) PS C:\Project\autodiff> git stash
Saved working directory and index state WIP on master: 1409930 feat: change main.py
(base) PS C:\Project\autodiff> git stash list
stash@{0}: WIP on master: 1409930 feat: change main.py
```

在我们做完了其他工作，想要恢复暂存的修改的时候，就可以执行 `git stash apply` 将修改恢复过来了。

```
(base) PS C:\Project\autodiff> git stash apply
On branch master
Changes not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
 modified: lab4-report-201300035.md

Untracked files:
 (use "git add <file>..." to include in what will be committed)
 images/2022-10-31-22-42-13.png
 images/2022-10-31-22-44-09.png
 images/2022-10-31-22-48-51.png
 images/2022-10-31-22-49-43.png
 images/2022-10-31-23-09-57.png

no changes added to commit (use "git add" and/or "git commit -a")
```

## 8. Git Checkout

执行 `git checkout -b dev` 创建 `dev` 分支，并使用 `git branch` 查看。

```
(base) PS C:\Project\autodiff> git checkout -b dev
Switched to a new branch 'dev'
(base) PS C:\Project\autodiff> git branch
* dev
 master
```

使用 `git checkout master` 和 `git checkout dev` 可以切换分支。

```
(base) PS C:\Project\autodiff> git checkout master
Switched to branch 'master'
M lab4-report-201300035.md
(base) PS C:\Project\autodiff> git checkout dev
Switched to branch 'dev'
M lab4-report-201300035.md
```

## 9. Git Merge

我们修改 `main.py` 的内容为，并 Commit 到 `dev` 分支上：

```
print('Hello World')
print('Hello Software Engineering')
print('New Branch')
```

```
(base) PS C:\Project\autodiff> git add *
(base) PS C:\Project\autodiff> git commit -m "test: test branch function"
[dev fa05ee3] test: test branch function
15 files changed, 86 insertions(+), 1 deletion(-)
```

我们执行 `git checkout master` 后执行 `git merge dev` 进行合并。

```
(base) PS C:\Project\autodiff> git branch
* dev
 master
(base) PS C:\Project\autodiff> git checkout master
Switched to branch 'master'
(base) PS C:\Project\autodiff> git merge dev
Updating db3adbb..fa05ee3
Fast-forward
 images/2022-10-31-22-42-13.png | Bin 0 -> 24583 bytes
 images/2022-10-31-22-44-09.png | Bin 0 -> 34070 bytes
```

执行 `git log --graph --decorate --oneline --all` 可以看出中途 `dev` 分支被分离了出去，进行了一次 Commit 之后才被合并的。

```
(base) PS C:\Project\autodiff> git log --graph --decorate --oneline --all
* e501cbc (refs/stash) WIP on dev: fa05ee3 test: test branch function
|\
| * 09a4676 index on dev: fa05ee3 test: test branch function
|/
* fa05ee3 (HEAD -> master, dev) test: test branch function
* db3adbb Revert "Revert "feat: change main.py""
* 800357d Revert "feat: change main.py"
* 1409930 feat: change main.py
* 1ad1f9c feat: init
```

## 10. Git Rebase

除了 Git Merge 命令之外，还有一种可以对分支进行合并的命令，即 Git Rebase 命令。

要理解 Git Rebase 命令其实很简单，我们知道每个版本其实保存了一个类似于父节点的指针，通过父节点指针我们就可以从 HEAD 一路找到初始化版本，并通过 Git Log 显示出来。理论上来说这个父节点指针不应该由我们控制，应该由 Git 自己管理，但是执行 Git Rebase 操作可以让我们更改分支的父节点指针，让当前分支的父节点指向另一个分支的最新节点，也就是“变基”操作。

我们先在 master 分支对 `main.py` 进行一点修改：

```
print('Hello World')
print('Hello Software Engineering')
print('New Branch')
print('Change For Branch Master')
```

然后执行 `git add *` 和 `git commit -m "fix: change for branch master"` 进行 Commit。

执行 `git checkout dev` 切换到 `dev` 分支, 并且同样在 `master` 分支对 `main.py` 进行一点修改:

```
print('Hello World')
print('Hello Software Engineering')
print('New Branch')
print('Change For Branch Dev')
```

然后执行 `git add *` 和 `git commit -m "fix: change for branch dev"` 进行 Commit。

此时我们执行 `git log --graph --decorate --oneline --all` 看一下情况:

```
(base) PS C:\Project\autodiff> git log --graph --decorate --oneline --all
* f71d701 (HEAD -> dev) fix: change for branch dev
| * a937151 (refs/stash) WIP on master: 629e4e9 fix: change for branch master
| | \
| | * 81b020e index on master: 629e4e9 fix: change for branch master
| | /
| * 629e4e9 (master) fix: change for branch master
| /
* fa05ee3 test: test branch function
* db3adbb Revert "Revert "feat: change main.py""
* 800357d Revert "feat: change main.py"
* 1409930 feat: change main.py
* 1ad1f9c feat: init
```

最关键的一步来了, 我们执行 `git rebase master` 进行变基, 将 `dev` 的父节点设为 `master` 的最新节点。

```
(base) PS C:\Project\autodiff> git rebase master
Auto-merging lab4-report-201300035.md
CONFLICT (content): Merge conflict in lab4-report-201300035.md
Auto-merging main.py
CONFLICT (content): Merge conflict in main.py
error: could not apply f71d701... fix: change for branch dev
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply f71d701... fix: change for branch dev
(base) PS C:\Project\autodiff> git stash apply
lab4-report-201300035.md: needs merge
main.py: needs merge
```

我们再次执行 `git log --graph --decorate --oneline --all` 看一下情况:

```
(base) PS C:\Project\autodiff> git log --graph --decorate --oneline --all
* ma 20f3db5 (refs/stash) WIP on dev: f71d701 fix: change for branch dev
| \
| * a9f620e index on dev: f71d701 fix: change for branch dev
| /
* f71d701 (dev) fix: change for branch dev
| * 629e4e9 (HEAD, master) fix: change for branch master
| /
* fa05ee3 test: test branch function
* db3adbb Revert "Revert "feat: change main.py""
* 800357d Revert "feat: change main.py"
* 1409930 feat: change main.py
* 1ad1f9c feat: init
```

最后我们执行 `git checkout master` 和 `git merge dev` 让 master 分支行进到最新的节点。

```
(base) PS C:\Project\autodiff> git checkout master
Switched to branch 'master'
(base) PS C:\Project\autodiff> git merge dev
Updating 629e4e9..fe4c32f
Fast-forward
 images/2022-11-01-10-33-28.png | Bin 0 -> 48594 bytes
 images/2022-11-01-10-39-34.png | Bin 0 -> 51341 bytes
 lab4-report-201300035.md | 22 ++++++++
 main.py | 3 ++-
 4 files changed, 23 insertions(+), 2 deletions(-)
 create mode 100644 images/2022-11-01-10-33-28.png
 create mode 100644 images/2022-11-01-10-39-34.png
```

## 11. Git Cherry Pick

对于多分支的代码库，将代码从一个分支转移到另一个分支是常见需求。这时分两种情况。一种情况是，你需要另一个分支的所有代码变动，那么就采用合并（`git merge`）。另一种情况是，你只需要部分代码变动（某几个提交），这时可以采用 Cherry Pick。

因此我们可以知道，Git Cherry Pick 命令很类似 Git Merge 命令，只不过可以选择合并某个 Commit 而不是合并某个分支。

我们执行 `git checkout dev` 切换到 dev 分支后，添加两个 Commit。

可以看出，我们当前有两个新的 Commit： `feat: change 1` 和 `feat: change 2`。

```
(base) PS C:\Project\autodiff> git log --graph --decorate --oneline --all
* 81b4b19 (HEAD -> dev) feat: change 2
* 40fad27 feat: change 1
| * 2b0cad4 (refs/stash) WIP on master: fe4c32f fix: change for branch dev
|/|
| * 62795de index on master: fe4c32f fix: change for branch dev
|/
* fe4c32f (master) fix: change for branch dev
* 629e4e9 fix: change for branch master
* fa05ee3 test: test branch function
* db3adbb Revert "Revert "feat: change main.py""
* 800357d Revert "feat: change main.py"
* 1409930 feat: change main.py
* 1ad1f9c feat: init
```

我们执行 `git checkout master` 切换到 master 分支后，然后执行

`git cherry-pick 40fad27` 只合并 dev 分支的 `feat: change 1`。

```
(base) PS C:\Project\autodiff> git log --graph --decorate --oneline --all
* 64e5807 (HEAD -> master) feat: change 1
| * 0d2f779 (refs/stash) WIP on dev: 81b4b19 feat: change 2
| |\
| | * 67b78ff index on dev: 81b4b19 feat: change 2
| |/
| * 81b4b19 (dev) feat: change 2
| * 40fad27 feat: change 1
|/
* fe4c32f fix: change for branch dev
* 629e4e9 fix: change for branch master
* fa05ee3 test: test branch function
* db3adbb Revert "Revert "feat: change main.py""
* 800357d Revert "feat: change main.py"
* 1409930 feat: change main.py
* 1ad1f9c feat: init
```

可以看到，我们的 master 分支确实合并了 `feat: change 1` Commit。

## 12. GitHub 远程仓库


我们先在 GitHub 上创建一个新的 Repo。



## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*

 OrangeX4 ▾

Repository name \*

/ autodiff ✓

Great repository names are short and memorable. Need inspiration? How about [animated-dollop](#)?

Description (optional)

autodiff - software engineering project

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

## 执行

```
git remote add origin https://github.com/OrangeX4/autodiff.git
git push -u origin master
git push origin dev
```

将所有分支推送上去。

```
(base) PS C:\Project\autodiff> git remote add origin https://github.com/OrangeX4/autodiff.git
(base) PS C:\Project\autodiff> git push -u origin master
Enumerating objects: 93, done.
Counting objects: 100% (93/93), done.
Delta compression using up to 8 threads
Compressing objects: 100% (84/84), done.
Writing objects: 100% (93/93), 900.35 KiB | 18.01 MiB/s, done.
Total 93 (delta 23), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (23/23), done.
To https://github.com/OrangeX4/autodiff.git
 * [new branch] master -> master
branch 'master' set up to track 'origin/master'.
```

可以看见在地址 <https://github.com/OrangeX4/autodiff/tree/master> 已经成功推送了上去。

dev had recent pushes 1 minute ago [Compare & pull request](#)

master ▾


2 branches

0 tags


[Go to file](#)

[Add file ▾](#)

[Code ▾](#)

 OrangeX4 docs: complete part of git 22e053b 12 minutes ago 9 commits

|            |                            |                |
|------------|----------------------------|----------------|
| cluster    | feat: init                 | 13 hours ago   |
| data/input | feat: init                 | 13 hours ago   |
| diff       | feat: init                 | 13 hours ago   |
| generator  | feat: init                 | 13 hours ago   |
| images     | docs: complete part of git | 12 minutes ago |
| input      | feat: init                 | 13 hours ago   |

**About** 

autodiff - software engineering project

☆ 0 stars

👁 1 watching

🍴 0 forks

**Releases**

No releases published

[Create a new release](#)

**Packages**

No packages published

[Publish your first package](#)

## 13. 分支合并图

这次实验中，我一共使用了 8 个分支，每个分支对应一个模块，而 master 是主分支。

```
(base) PS C:\Project\autodiff> git branch
 cluster
 dev
 diff
 generator
 input
* master
 output
 paracomp
```

使用 `git log --graph --decorate --oneline --all` 展示的分支合并图如下：

```
(base) PS C:\Project\autodiff> git log --graph --decorate --oneline --all
* 804c7f0 (HEAD -> master, output) docs: complete output
* 7360bf6 feat: complete output
* 933b4d5 (paracomp) feat: add clean for paracomp
* 70f8df8 (diff) feat: add clean_cmd for diff
* b5ed014 docs: complete paracomp part
* fa5dc56 fix: fix paracomp
* a1013d0 feat: complete paracomp
| * c88463c (refs/stash) WIP on diff: ce75689 Merge branch 'generator'
|/|
| * 1230f40 index on diff: ce75689 Merge branch 'generator'
|/
* ce75689 Merge branch 'generator'
|\
| * 8547816 (generator) feat: update generator
* | 7ed9fe1 (input) docs: complete input part
* | c9cfe2f feat: complete input
* | f80f9be (cluster) docs: complete cluster part
* | cb7cf3b feat: complete cluster
|/
* 8566c66 (origin/master) docs: complete generator part
* a2ef204 feat: complete generator
* 87b17d0 docs: complete diff part
* fdd264f feat: add more languages
* f488b74 feat: complete diff
* 3557b52 docs: add github
* 22e053b docs: complete part of git
* 64e5807 feat: change 1
| * 81b4b19 (origin/dev, dev) feat: change 2
| * 40fad27 feat: change 1
|/
* fe4c32f fix: change for branch dev
* 629e4e9 fix: change for branch master
* fa05ee3 test: test branch function
* db3adbb Revert "Revert "feat: change main.py""
* 800357d Revert "feat: change main.py"
* 1409930 feat: change main.py
* 1ad1f9c feat: init
```

## 二、代码架构



# 1. 执行比较 (diff)

diff 模块是执行比较的核心模块，在这里我们使用 Python 中 subprocess 模块的 Popen 函数实现。

想要进行比较，我们必须先将源代码文件编译为可执行文件，再执行可执行文件，并向其输入我们生成的输入文本，然后获取输出结果。

我编写了一个类 `Executor`，由其来负责「编译」和「执行」两个功能。

```
class Executor:

 def __init__(self, execute_cmd: str, build_cmd=None, clean_cmd=None) -> None:
 """
 execute_cmd: 用于执行可执行程序的命令
 build_cmd: 用于构建可执行程序的命令
 clean_cmd: 用于清理可执行程序的命令
 """
 self.execute_cmd = execute_cmd
 self.build_cmd = build_cmd
 self.clean_cmd = clean_cmd

 def _exec(self, file: str, cmd) -> None:
 if cmd is None:
 return
 cmd = format_string_with_file(cmd, file)
 process = Popen(cmd, stdout=PIPE, stderr=PIPE)
 process.communicate()

 def build(self, file: str) -> None:
 self._exec(file, self.build_cmd)

 def clean(self, file: str) -> None:
 self._exec(file, self.clean_cmd)

 def execute(self, file: str, input: str) -> str:
 cmd = format_string_with_file(self.execute_cmd, file)
 process = Popen(cmd, stdout=PIPE, stderr=PIPE, stdin=PIPE)
 output, err = process.communicate(input=input.encode())
 if err:
 # raise Exception(err.decode())
 return err.decode()
 return output.decode()
```

其中我们通过 `process = Popen(cmd)` 来打开可执行程序，然后使用 `process.communicate(input)` 向可执行程序输入我们生成的输入文本，最后获取到输出 `output`。

借助这个 `Executor` 类，我们就可以用很具有扩展性的方式，为不同平台，乃至不同语言，注册不同的执行器。

```

self.executor_map = {
 'c': {
 'Windows': Executor(
 '{fileNoExtension}.exe',
 'gcc {file} -o "{fileNoExtension}.exe"',
 'del "{fileNoExtension}.exe"'),
 'Linux': Executor(
 './{fileNoExtension}.out',
 'gcc {file} -o "{fileNoExtension}.out"',
 'rm "{fileNoExtension}.out"')
 },
 'cpp': {
 'Windows': Executor(
 '{fileNoExtension}.exe',
 'g++ "{file}" -o "{fileNoExtension}.exe"',
 'del "{fileNoExtension}.exe"'),
 'Linux': Executor(
 '{fileNoExtension}.out',
 'g++ "{file}" -o "{fileNoExtension}.out"',
 'rm "{fileNoExtension}.out"')
 },
 'py': {
 'Windows': Executor('python "{file}"'),
 'Linux': Executor('python3 "{file}"')
 },
}
}

```

同时我也写了一个单元测试，单元测试的输出结果为：

```

input: 2
output: {'../data/input/4A/48762087.cpp': 'HELLO',
 '../data/input/4A/84822638.cpp': 'NO\r\n'}
result: False

```

说明该模块能够正常工作了。

## 2. 生成样例 (generator)

为了实现样例生成器，也即是 将 `int(a, b)`，`char`，`string(a, b)` 替换为相应的随机值，这里我采用了正则表达式的方式实现。

核心代码为：

```

def generate(self, input: str) -> str:
 ...

 int(a, b): a <= value(int) <= b
 char: 随机大小写字母
 string(a, b): 由 char 组成, a <= length(string) <= b
 ...

 def get_char():
 return chr(randint(97, 122) if randint(0, 1) else randint(65, 90))

 # 将 int(a, b), char, string(a, b) 替换为对应的值
 input = re.sub(r'int\\(\\s*(\\d+)\\s*,\\s*(\\d+)\\s*\\)',
 lambda x: str(randint(int(x.group(1)), int(x.group(2)))), input)
 input = re.sub(r'char', lambda x: get_char(), input)
 input = re.sub(r'string\\(\\s*(\\d+)\\s*,\\s*(\\d+)\\s*\\)',
 lambda x: ''.join([get_char() for _ in
 range(randint(int(x.group(1)), int(x.group(2)))]), input)
 return input

```

即是使用 Python 的 `re.sub(pattern, func, text)` 方法来实现匹配和替换。

### 3. 中间表示与等价类 (cluster)

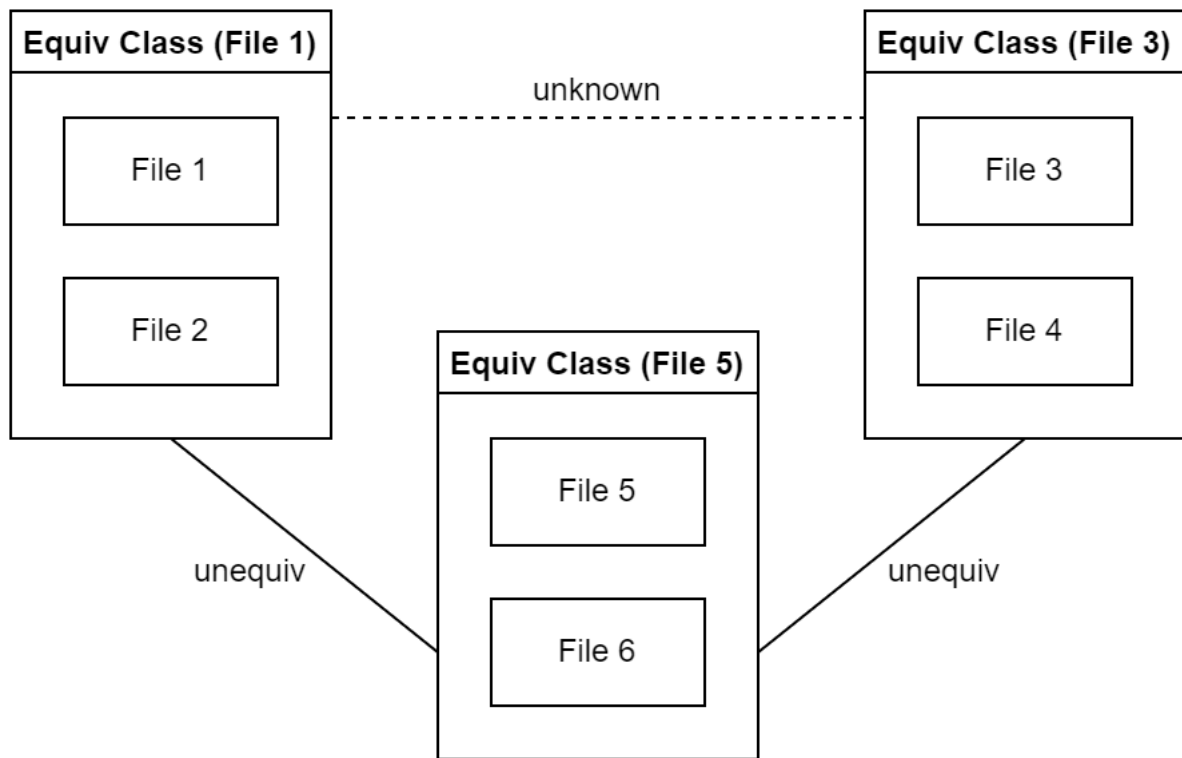
我们使用一个中间表示类 Cluster, 来表示我们当前处理的数据的状态。

```

{
 "cluster_name": "4A",
 "random_input_generator": {
 "type": "stdin_format.txt",
 "content": "int(1, 3)"
 },
 "custom_input": ["1", "2"],
 "config": {
 "random_test_times": 10,
 "random_seed": 0,
 },
 "files": {
 "48762087.cpp": {
 "content": "int main() { return 0; }",
 "equiv_class": "48762087.cpp"
 },
 "84822638.cpp": {
 "content": "int main() { return 0; }",
 "equiv_class": "48762087.cpp"
 }
 },
 "equiv": [["48762087.cpp", "84822638.cpp"]],
 "unequiv": [],
 "diff": {
 "48762087.cpp": {
 "84822638.cpp": {
 "auto": "unknown",
 "manual": "equiv",
 "logic": "equiv"
 }
 },
 "84822638.cpp": {
 "48762087.cpp": {
 "auto": "unknown",
 "manual": "equiv",
 "logic": "equiv"
 }
 }
 }
}

```

其中 `files` 属性下的 `equiv_class` 用的是一个 **并查集结构**，我们用并查集来标识该文件所属的 **等价类**。



如图所示：

- 在同一个等价类里的文件被认为是 **相互等价** 的，通过一个根文件来标识等价类，例如下面的那个等价类通过 `File 5` 标识。
- 在不同等价类之间，通过实线连接的等价类被认为是 **不等价** 的，即两个等价类里的文件两两匹配都不等价。
- 在不同等价类之间，通过虚线连接的等价类被认为是 **未知等价关系** 的，需要用户进一步地判断 (虚线是初始化后就存在的)。

借助这幅图所示的等价类概念，以及并查集的算法知识，我们就可以写出一个能够让用户 **手动动态更改等价关系** 的 Cluster 类。

由于代码有数百行，就不在报告中过多展示，感兴趣可以查阅代码 `cluster/__init__.py`。

代码还包括了一个简易的单元测试：

```
cluster = Cluster('test', cluster)
cluster.clear()
cluster.set_manual('1', '2', 'equiv')
cluster.set_manual('2', '3', 'equiv')
cluster.set_manual('2', '3', 'unequiv')
cluster.set_manual('2', '3', 'unknown')
cluster.set_manual('3', '4', 'equiv')
cluster.set_auto('2', '3', 'unequiv')
cluster.update_diff()
print(cluster.cluster)
```

可以看出，接口是很易用的，可以在后续等价确认工具中使用。

## 4. 输入 (input)

输入模块负责把 `data/input` 里的文件读入，大部分都是繁杂的读入代码，这里就不再赘述。

代码中值得称道的一部分有，可以高拓展性地支持其他配置文件，除了 `stdin_format.txt` 外，我还添加了 `stdin_format.py` 和 `config.json` 等额外的配置文件，以及自定义测试样例，可以进行更灵活的随机样例生成和配置。

```
with open(file_path, 'r', encoding='utf-8') as f:
 content = f.read()
 # 各种配置文件
 if file_name == 'stdin_format.txt':
 cluster['random_input_generator']['type'] = 'stdin_format.txt'
 cluster['random_input_generator']['content'] = content
 elif file_name == 'stdin_format.py':
 cluster['random_input_generator']['type'] = 'stdin_format.py'
 elif file_name == 'config.json':
 cluster['config'].update(json.loads(content))
 else:
 # 普通文件
 cluster['files'][file_name] = {
 'content': content,
 'equiv_class': file_name
 }
```

```
if file_name == 'custom_input':
 # 自定义输入
 for custom_input_file_name in os.listdir(file_path):
 custom_input_file_path = os.path.join(
 file_path, custom_input_file_name)
 with open(custom_input_file_path, 'r', encoding='utf-8') as f:
 cluster['custom_input'].append(f.read())
```

## 5. 并行多进程计算 (paracomp)

为了实现并行地进行「文件编译」和「文件比对」，同时也因为 Python 的 GIL 机制导致无法正常多线程计算，这里我采用了 **多进程** 与 **进程池** 的策略来实现并行计算。

例如，在文件编译过程中，我使用了如下逻辑使得 `g++ xxx.cpp` 这个过程能够并行地进行：

```

生成所有可执行文件
build_pool = multiprocessing.Pool(self.proc_num)
for file in cluster_dict['files']:
 file_path = os.path.abspath(os.path.join(
 self.path, 'input', cluster_name, file))
 build_pool.apply_async(self.diff.build, args=(file_path,))
build_pool.close()
build_pool.join()

```

而在文件比对部分，我也同样采取多进程的方式，进行样例生成与文件比对，大致代码如下：

```

对文件对进行比较
diff_pool = multiprocessing.Pool(self.proc_num)
for file1 in cluster_dict["files"]:
 result[file1] = {}
 for file2 in cluster_dict["files"]:
 # 判断字符串大小，保证不重复
 if file1 >= file2:
 continue
 # 为进程池加入任务
 result[file1][file2] = diff_pool.apply_async(
 self.diff_func, args=(cluster_name, file1, file2))
diff_pool.close()
diff_pool.join()
获取执行结果
for file1 in result:
 for file2 in result[file1]:
 res = result[file1][file2].get()
 cluster.set_auto(file1, file2, 'equiv' if res else 'unequiv')
cluster.update_diff()

```

在对比结束后，我们还会对可执行文件进行清理：

```

清除可执行程序
for file in cluster_dict['files']:
 file_path = os.path.abspath(os.path.join(
 self.path, 'input', cluster_name, file))
 self.diff.clean(file_path)

```

最后，我同样写了一个单元测试进行验证：

```

def unit_test():
 paracomp = Paracomp('../data', Input('../data'))
 print(paracomp.get_cluster_names())
 paracomp.run('4A')
 print(json.dumps(paracomp.input.clusters['4A'].cluster))

```

最后验证结果为多进程并行模块能够完美地发挥作用，由于我的电脑是 8 核处理器，我设定了进程池最大为 8 个进程，因此执行速度是原来的 8 倍。

其中一部分的输出结果是：

```
"127473352.cpp": {
 "101036360.cpp": {
 "auto": "unequiv",
 "manual": "unknown",
 "logic": "unequiv"
 },
 "117364748.cpp": {
 "auto": "unequiv",
 "manual": "unknown",
 "logic": "unequiv"
 },
 "134841308.cpp": {
 "auto": "equiv",
 "manual": "unknown",
 "logic": "unknown"
 }
}
```

## 6. 输出 (output)

输出部分也比较简单，就是繁杂的代码，其中输出 CSV 文件的部分如下：

```
def save_diff_list_to_csv(self) -> None:
 ...
 保存到 path/output/equal.csv 和 path/output/inequal.csv
 格式为 input/cluster_name/file1,input/cluster_name/file2
 其中是否 equal 只取决于 auto == 'equiv' 与 auto == 'unequiv'
 ...

 diff_list = self.clusters_to_diff_list()
 equal_csv_path = os.path.abspath(
 os.path.join(self.path, "output", "equal.csv"))
 inequal_csv_path = os.path.abspath(
 os.path.join(self.path, "output", "inequal.csv"))
 equal_csv_content = "file1,file2"
 inequal_csv_content = "file1,file2"
 for diff in diff_list:
 _file1 = f'input/{diff["cluster_name"]}/{diff["file1"]}'
 _file2 = f'input/{diff["cluster_name"]}/{diff["file2"]}'
 if diff["auto"] == "equiv":
 equal_csv_content += f'\n{_file1},{_file2}'
 elif diff["auto"] == "unequiv":
 inequal_csv_content += f'\n{_file1},{_file2}'
 with open(equal_csv_path, 'w') as f:
 f.write(equal_csv_content)
 with open(inequal_csv_path, 'w') as f:
 f.write(inequal_csv_content)
```



值得一提的是，我还完成了一个保存 clusters 的 json 中间文件的方法，这使得我们可以将计算好的结果保存起来，以供以后使用，下次加载的时候就不必从头开始计算了。

## 三、运行流程

第一步，需要先将项目 clone 下来。

```
git clone https://github.com/OrangeX4/autodiff.git
```

第二步，安装 Python 开发环境。

第三步，往 `data/input` 里加入数据，例如 `4A/xxx.cpp`、`50A/xxx.cpp` 等。

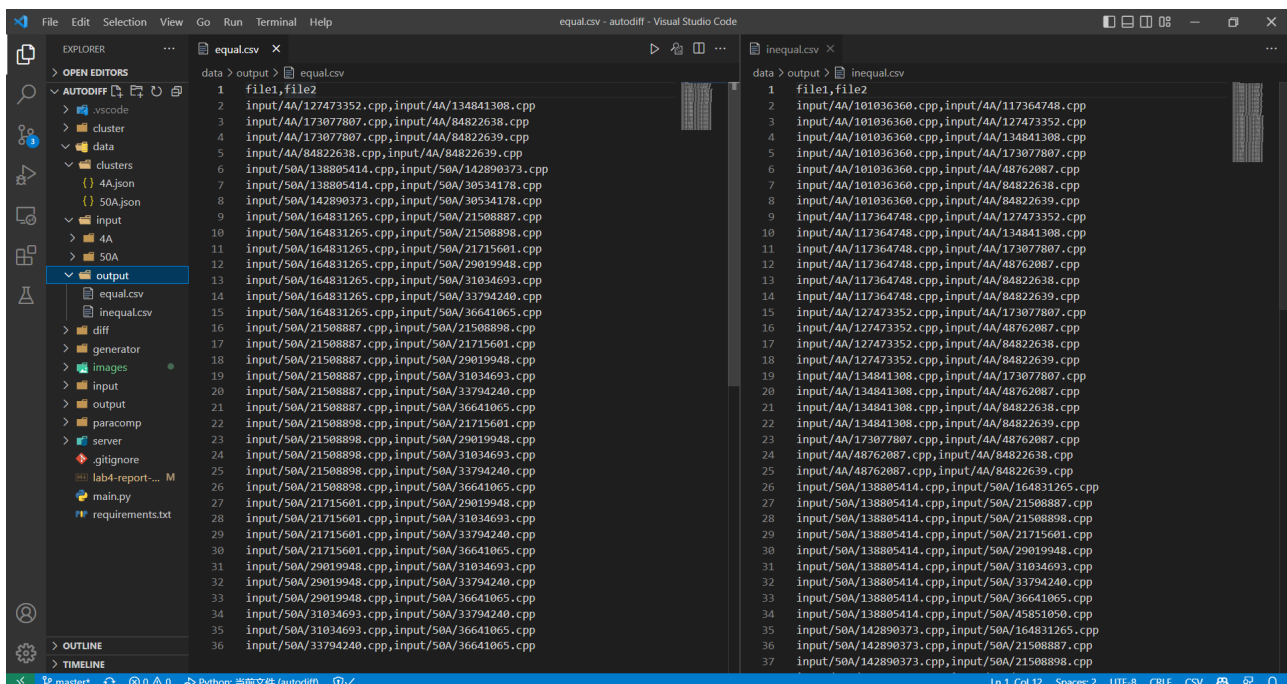
第四步，执行 `python main.py`，其中 `main.py` 的代码为：

```
def main(path: str):
 # 读取输入，from_clusters 表示会读取保存的 clusters
 print('读取输入中...')
 input = Input(path, from_clusters=True)
 # 进行并行比较
 print('执行比较中...')
 paracomp = Paracomp(path, input)
 for cluster_name in paracomp.get_cluster_names():
 # 如果是加载的则跳过
 if input.clusters[cluster_name].cluster['is_loaded']:
 continue
 paracomp.run(cluster_name)
 # 对结果进行保存
 output = Output(path, input.clusters)
 print('保存 csv 文件中...')
 output.save_diff_list_to_csv()
 # 同时也保存 clusters 到 clusters 文件夹
 print('保存 clusters 文件中...')
 output.save_clusters()
 print('完成!')
```

可见，每一步都十分清晰，通过输入模块读取位于 `data/input` 的输入，再通过 `paracomp.run(cluster_name)` 进行多进程并发比较，最后保存到 CSV 文件中。

最后 `data` 文件夹的文件树：

```
├clusters
│ 4A.json
│ 50A.json
├input
└output
 equal.csv
 inequal.csv
```



```
equal.csv
1 file1,file2
2 input/4A/127473352.cpp,input/4A/134841308.cpp
3 input/4A/173077807.cpp,input/4A/84822638.cpp
4 input/4A/173077807.cpp,input/4A/84822639.cpp
5 input/4A/84822638.cpp,input/4A/84822639.cpp
6 input/50A/138805414.cpp,input/50A/142890373.cpp
7 input/50A/138805414.cpp,input/50A/30534178.cpp
8 input/50A/142890373.cpp,input/50A/30534178.cpp
9 input/50A/164831265.cpp,input/50A/21508887.cpp
10 input/50A/164831265.cpp,input/50A/21508898.cpp
11 input/50A/164831265.cpp,input/50A/21715601.cpp
12 input/50A/164831265.cpp,input/50A/29019948.cpp
13 input/50A/164831265.cpp,input/50A/31034693.cpp
14 input/50A/164831265.cpp,input/50A/33794240.cpp
15 input/50A/164831265.cpp,input/50A/36641065.cpp
16 input/50A/21508887.cpp,input/50A/21508898.cpp
17 input/50A/21508887.cpp,input/50A/21715601.cpp
18 input/50A/21508887.cpp,input/50A/29019948.cpp
19 input/50A/21508887.cpp,input/50A/31034693.cpp
20 input/50A/21508887.cpp,input/50A/33794240.cpp
21 input/50A/21508887.cpp,input/50A/36641065.cpp
22 input/50A/21508898.cpp,input/50A/21715601.cpp
23 input/50A/21508898.cpp,input/50A/29019948.cpp
24 input/50A/21508898.cpp,input/50A/31034693.cpp
25 input/50A/21508898.cpp,input/50A/33794240.cpp
26 input/50A/21508898.cpp,input/50A/36641065.cpp
27 input/50A/21715601.cpp,input/50A/29019948.cpp
28 input/50A/21715601.cpp,input/50A/31034693.cpp
29 input/50A/21715601.cpp,input/50A/33794240.cpp
30 input/50A/21715601.cpp,input/50A/36641065.cpp
31 input/50A/29019948.cpp,input/50A/31034693.cpp
32 input/50A/29019948.cpp,input/50A/33794240.cpp
33 input/50A/29019948.cpp,input/50A/36641065.cpp
34 input/50A/31034693.cpp,input/50A/33794240.cpp
35 input/50A/31034693.cpp,input/50A/36641065.cpp
36 input/50A/33794240.cpp,input/50A/36641065.cpp

inequal.csv
1 file1,file2
2 input/4A/101036360.cpp,input/4A/117364748.cpp
3 input/4A/101036360.cpp,input/4A/127473352.cpp
4 input/4A/101036360.cpp,input/4A/134841308.cpp
5 input/4A/101036360.cpp,input/4A/173077807.cpp
6 input/4A/101036360.cpp,input/4A/48762087.cpp
7 input/4A/101036360.cpp,input/4A/84822638.cpp
8 input/4A/101036360.cpp,input/4A/84822639.cpp
9 input/4A/117364748.cpp,input/4A/127473352.cpp
10 input/4A/117364748.cpp,input/4A/134841308.cpp
11 input/4A/117364748.cpp,input/4A/173077807.cpp
12 input/4A/117364748.cpp,input/4A/48762087.cpp
13 input/4A/117364748.cpp,input/4A/84822639.cpp
14 input/4A/117364748.cpp,input/4A/84822639.cpp
15 input/4A/127473352.cpp,input/4A/173077807.cpp
16 input/4A/127473352.cpp,input/4A/48762087.cpp
17 input/4A/127473352.cpp,input/4A/84822638.cpp
18 input/4A/127473352.cpp,input/4A/84822639.cpp
19 input/4A/134841308.cpp,input/4A/173077807.cpp
20 input/4A/134841308.cpp,input/4A/48762087.cpp
21 input/4A/134841308.cpp,input/4A/84822639.cpp
22 input/4A/134841308.cpp,input/4A/84822639.cpp
23 input/4A/173077807.cpp,input/4A/48762087.cpp
24 input/4A/48762087.cpp,input/4A/84822638.cpp
25 input/4A/48762087.cpp,input/4A/84822639.cpp
26 input/50A/138805414.cpp,input/50A/164831265.cpp
27 input/50A/138805414.cpp,input/50A/21508887.cpp
28 input/50A/138805414.cpp,input/50A/21508898.cpp
29 input/50A/138805414.cpp,input/50A/21715601.cpp
30 input/50A/138805414.cpp,input/50A/29019948.cpp
31 input/50A/138805414.cpp,input/50A/31034693.cpp
32 input/50A/138805414.cpp,input/50A/33794240.cpp
33 input/50A/138805414.cpp,input/50A/36641065.cpp
34 input/50A/138805414.cpp,input/50A/45851050.cpp
35 input/50A/142890373.cpp,input/50A/164831265.cpp
36 input/50A/142890373.cpp,input/50A/21508887.cpp
37 input/50A/142890373.cpp,input/50A/21508898.cpp
```

## 四、优秀设计

### 1. 执行模块的扩展性

借助 `diff` 模块中的 `Executor` 类，我们就可以用很具有扩展性的方式，为 **不同平台**，乃至 **不同语言**，注册不同的执行器。

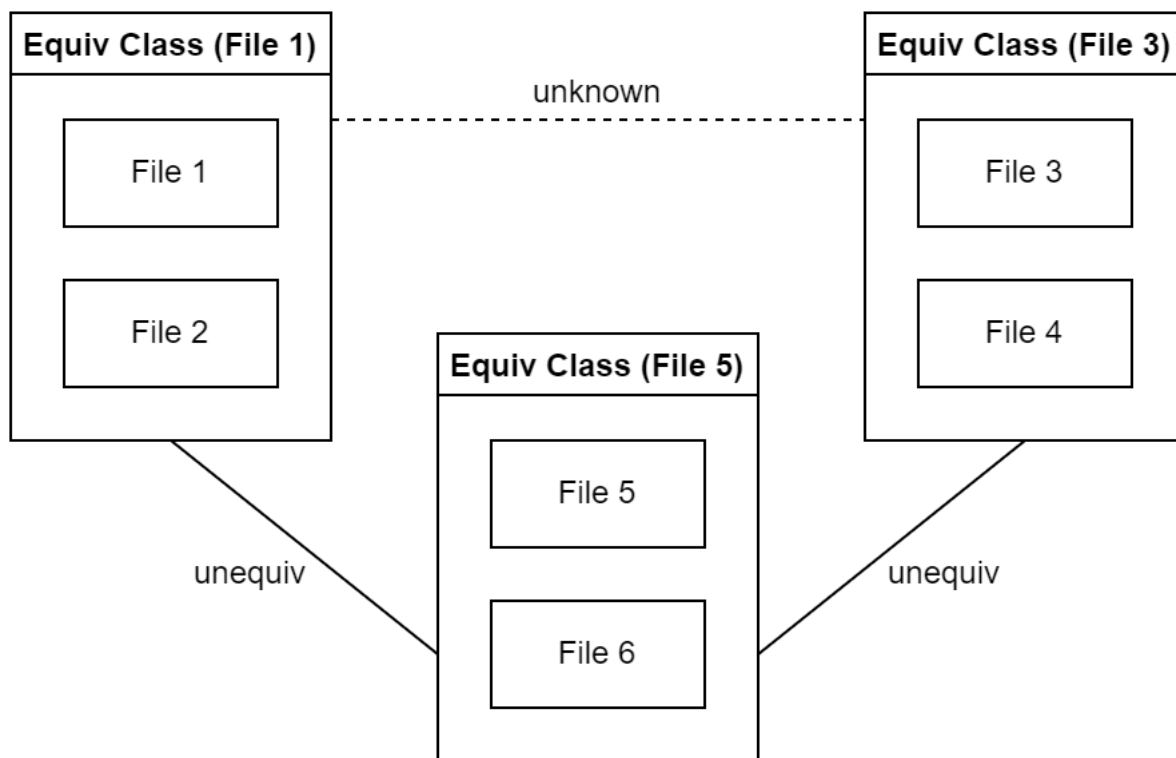
```

self.executor_map = {
 'c': {
 'Windows': Executor(
 '{fileNoExtension}.exe',
 'gcc {file} -o "{fileNoExtension}.exe"',
 'del "{fileNoExtension}.exe"'),
 'Linux': Executor(
 './{fileNoExtension}.out',
 'gcc {file} -o "{fileNoExtension}.out"',
 'rm "{fileNoExtension}.out"')
 },
 'cpp': {
 'Windows': Executor(
 '{fileNoExtension}.exe',
 'g++ "{file}" -o "{fileNoExtension}.exe"',
 'del "{fileNoExtension}.exe"'),
 'Linux': Executor(
 '{fileNoExtension}.out',
 'g++ "{file}" -o "{fileNoExtension}.out"',
 'rm "{fileNoExtension}.out"')
 },
 'py': {
 'Windows': Executor('python "{file}"'),
 'Linux': Executor('python3 "{file}"')
 },
}
}

```

## 2. 基于并查集的等价类

我们使用 `cluster` 模块 `files` 属性下的 `equiv_class` 的一个 **并查集结构** 来标识该文件所属的 **等价类**。



如图所示：

- 在同一个等价类里的文件被认为是 **相互等价** 的，通过一个根文件来标识等价类，例如下面的那个等价类通过 `File 5` 标识。
- 在不同等价类之间，通过实线连接的等价类被认为是 **不等价** 的，即两个等价类里的文件两两匹配都不等价。
- 在不同等价类之间，通过虚线连接的等价类被认为是 **未知等价关系** 的，需要用户进一步地判断 (虚线是初始化后就存在的)。

借助这幅图所示的等价类概念，以及并查集的算法知识，我们就可以写出一个能够让用户 **手动动态更改等价关系** 的 Cluster 类。

### 3. 测试样例的多样性

除了 `stdin_format.txt` 外，我还添加了 `stdin_format.py` 和 `config.json` 等额外的配置文件，以及自定义测试样例，可以进行更灵活的随机样例生成和配置。

- `stdin_format.txt`：使用正则表达式实现的文本替换与随机生成。
- `stdin_format.py`：通过用户自定义 Python 程序来实现更复杂的 (如数组、矩阵) 的输入样例生成。
- `config.json`：可以配置 **生成随机样例的个数** 以及 **随机数种子** 等配置。

### 4. 并行多进程计算

并行地进行「文件编译」和「文件比对」，并且采用了 **多进程** 与 **进程池** 的策略来实现并行计算。

设定了进程池最大为 8 个进程，因此执行速度是原来的 8 倍。

## 5. 保存中间状态

保存 clusters 的 json 中间文件的方法，这使得我们可以将计算好的结果保存起来，以供以后使用，下次加载的时候就不必从头开始计算了。

## 6. 为后续的前端界面预留了接口

我已经实现了后续等价确认工具的前端 UI：



我在本次实验中预留了很多接口，例如 `def clusters_to_diff_list(self) -> list:` 接口，可供下一次实验实现等价确认工具预留了很多有用的接口。