中文版本（机翻）附于末尾

Project Code File Structure

The project is written in C++ and employs relatively abstract object-oriented encapsulation, which may make it somewhat obscure.

Even though the project is small, it uses CMake for management.

CMakeLists.txt

`CMakeLists.txt` is the core configuration file for the CMake build system, defining project structure, compilation options, dependencies, and cross-platform build rules to generate native build environments (e.g., Makefiles or Visual Studio projects).

Graph.cpp / Graph.h

This module provides functionality for representing and analyzing graphs. It includes a `Graph` class to store graph data (nodes and weighted edges) and a `GraphShortestPathSolution` class to compute and count shortest paths from a given start node to any destination node in the graph. The implementation supports graphs with up to 500 nodes.

> Note: You may think the class `GraphShortestPathSolution` class is abstract, but in fact the encapsulation of `GraphShortestPathSolution` as a separate class from `Graph` follows the separation of concerns principle, where the graph structure (`Graph`) is decoupled from the algorithm logic (`GraphShortestPathSolution`). This design offers flexibility—multiple algorithms or solutions (e.g., Dijkstra, BFS) can operate on the same `Graph` instance without modifying its core structure. It also enables state isolation, as each solution (e.g., different start nodes) maintains its own `visited`, `path_parent`, and `path_length` arrays, avoiding redundant recomputation. While unintuitive at first, this approach improves modularity, reusability, and scalability for graph algorithms.

main.cpp/main(.exe)

This module provides the main entry point for the simplification program, handling input processing, graph analysis, and result output. It reads the graph structure (nodes, weighted edges) and test cases from standard input, then computes nodes that appear in shortest paths above a specified threshold.

Usage:

- `main < (Input Stream)`

test.cpp/test(.exe)

This module implements an automated testing framework for graph analysis, handling input/output validation, correctness checking, and performance measurement. It extends the main program with file-based testing capabilities and cross-platform compatibility.

Usage:

- `test <input_sample> <output_sample>`
- `<input_sample>`: Path to the test case input file containing graph data and queries
- `<output_sample>`: Path to the expected output file for correctness verification

Outputs:

- `Correct`/`Wrong` verdicts with execution time

Folder: test_sample

This directory includes some testcases for the project, which can be used in the test module.

samplegen.py

This is a python script to generate a large ramdom testcase. It does not need any arguments, but should be run with Python interpreter.

> I recommend that you redirect the output stream to a file.

README.md / README.txt / README.pdf

The README file, nothing to introduce.

How to Build and Run

> If you feel confused, you can skip to Binary Release

This project is organized with CMake, a cross-platform build system. Below are the steps to build and run the project on your local machine.

Prerequisites

- **CMake**
- **C Compiler**: A C compiler like `g++` or `clang++`.
- **ninja** or **make**

Building the Project

1. **Open terminal in the code folder**:
   Open terminal and change directory to the code folder
2. **Create a Build Directory**:
   Create a separate directory for building the project to keep the source tree clean.

   ```
   mkdir build
   cd build
   ```

3. **Generate Build Files**:
   Run CMake to generate the build files (e.g., Makefiles) in the build directory.

   ```
   cmake -G "Ninja" ..
   ```

   or

   ```
   cmake -G "Unix Makefiles" ..
   ```

   This command will read the `CMakeLists.txt` file in the parent directory and generate the necessary build files.

4. **Compile the Project**:
   Use the generated build files to compile the project. If you're using ninja, you can compile the project by running:

   ```
   ninja
   ```

   If you're using make, you can compile the project by running:

   ```
   make
   ```

   or (in Windows)

   ```
   mingw32-make
   ```

   This will compile the source code and generate the executables: `main` and `test`. or (in Windows) `main.exe` and `test.exe`.

Running the Executables

After successfully building the project, you can run the generated executables from the build directory.

1. **Run the Main Program**:

   ```
   ./main
   ```

2. **Run the Test Module**:

   ```
   
   ```

```
./test <input_sample> <output_sample>`
```

- `<input_sample>`: Path to the test case input file containing graph data and queries
- `<output_sample>`: Path to the expected output file for correctness verification

Cleaning the Build

If you want to clean the build and remove all the generated files, you can run:

```
ninja clean
```

or

```
make clean
```

or

```
mingw32-make clean
```

This will remove the compiled objects and executables, but keep the build files generated by CMake.

Rebuilding the Project

If you make changes to the source code, you can rebuild the project by running:

`minja` or `make` or `ming32-make`

from the build directory. CMake will automatically detect changes and recompile only the necessary files.

Additional CMake Options

- **Build Type**: You can specify the build type (e.g., Debug or Release) when generating the build files:

  ```
  cmake -DCMAKE_BUILD_TYPE=Debug ..
  ```

  or

  ```
  cmake -DCMAKE_BUILD_TYPE=Release ..
  ```

Binary Release

Binary releases for **Windows-amd64** and **Linux-amd64** are provided.

The function of the two files are mentioned in the first section, see above.

Remind that you shouldn't directly double-click the test module, but run them in terminal with arguments quoted by "".

You can rename the downloaded file by yourself.

Windows

Compiler: gcc version 14.2.0 (Rev3, Built by MSYS2 project)
Binaries:

- main.exe：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/uploads/2025/04/21/1809455497PiFKIRQO.exe
- test.exe：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/uploads/2025/04/21/1810126700ncKU3EXl.exe

Linux

Compiler: gcc version 12.2.0 (Debian 12.2.0-14)
Binaries:

- main：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/main_elf
- test：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/test_elf

Crash?

You may find that the program closes immediately after you finish entering the data. This happens because the program has already printed the result and completed all its execution logic, so it exits on its own.

To avoid missing the output, you should run the program in a terminal instead of double-clicking it. If you're not sure what "terminal" means, please copy this paragraph and ask an AI for help.

项目代码文件结构

本项目使用 C++ 编写，并采用了相对抽象的面向对象封装方式，可能会显得有些晦涩。

尽管项目体积较小，但它仍然使用 CMake 进行管理。

CMakeLists.txt

`CMakeLists.txt` 是 CMake 构建系统的核心配置文件，定义了项目结构、编译选项、依赖项和跨平台的构建规则，用于生成本地构建环境（如 Makefile 或 Visual Studio 工程）。

Graph.cpp / Graph.h

该模块提供用于表示和分析图的功能。它包含一个 `Graph` 类用于存储图数据（节点和带权边），以及一个 `GraphShortestPathSolution` 类，用于计算并计数从给定起点到图中任意目标点的所有最短路径。实现支持最多 500 个节点的图。

> 注意：你可能会认为 `GraphShortestPathSolution` 类是抽象的，但事实上，将 `GraphShortestPathSolution` 与 `Graph` 分离成两个类，是出于关注点分离原则的考虑：图结构（`Graph`）与算法逻辑（`GraphShortestPathSolution`）被解耦。这种设计提供了灵活性——多个算法或解法（如 Dijkstra、BFS）都可以在同一个 `Graph` 实例上运行，而无需修改其核心结构。同时也实现了状态隔离，每个解法（例如使用不同的起点）都有自己独立的 `visited`、`path_parent` 和 `path_length` 数组，避免重复计算。虽然初看上去不太直观，但这种方式提升了图算法模块的可维护性、可复用性和可扩展性。

main.cpp/main(.exe)

该模块提供了简化程序的主入口，处理输入解析、图分析以及结果输出。它从标准输入读取图结构（节点、带权边）和测试用例，然后计算在最短路径中出现频率超过特定阈值的节点。

使用方法:

- `main < (输入流)`

test.cpp/test(.exe)

该模块实现了图分析的自动化测试框架，处理输入/输出验证、正确性检查以及性能测量。它在主程序基础上扩展了基于文件的测试能力，并具有跨平台兼容性。

使用方法:

- `test <input_sample> <output_sample>`
- `<input_sample>`：包含图数据和查询的测试用例输入文件路径
- `<output_sample>`：用于验证正确性的预期输出文件路径

输出:

- 显示执行结果 `Correct` / `Wrong`，以及运行时间

文件夹：test_sample

该目录包含了一些项目的测试用例，可用于 test 模块中。

samplegen.py

这是一个用于生成大型随机测试用例的 Python 脚本。它不需要传入参数，但应由 Python 解释器运行。

> 建议你将输出流重定向到一个文件中。

README.md / README.txt / README.pdf

README 文件，无需介绍。

如何构建和运行

> 如果你感到困惑，可以直接跳到 二进制发布版本

本项目使用 CMake 构建系统进行组织。以下是在本地构建和运行项目的步骤。

先决条件

- **CMake**
- **C 编译器**：如 `g++` 或 `clang++`
- **ninja** 或 **make**

构建项目

1. **在代码目录中打开终端**：
   打开终端并切换到代码所在的目录

2. **创建构建目录**：
   创建一个单独的目录来构建项目，以保持源码目录整洁。

   ```
   mkdir build
   cd build
   ```

3. **生成构建文件**：
   运行 CMake 以在构建目录中生成构建文件（如 Makefile）。

   ```
   cmake -G "Ninja" ..
   ```

   或者

   ```
   cmake -G "Unix Makefiles" ..
   ```

   该命令会读取上级目录中的 `CMakeLists.txt` 文件，并生成必要的构建文件。

4. **编译项目**：
   使用生成的构建文件编译项目。
   如果你使用 ninja，可以运行：

   ```
   ninja
   ```

如果你使用 make，可以运行：

```
make
```

或（Windows 下）

```
mingw32-make
```

这将编译源码，并生成可执行文件：`main` 和 `test`。
或（Windows 下）
`main.exe` 和 `test.exe`。

运行可执行文件

在成功构建项目后，你可以从构建目录中运行生成的可执行文件。

1. **运行主程序**：

   ```
   ./main
   ```

2. **运行测试模块**：

   ```
   ./test <input_sample> <output_sample>
   ```

   - `<input_sample>`：包含图数据和查询的测试用例输入文件路径
   - `<output_sample>`：用于验证正确性的预期输出文件路径

清理构建

如果你想清理构建并移除所有生成的文件，可以运行：

```
ninja clean
```

或

```
make clean
```

或

```
mingw32-make clean
```

这将移除编译生成的对象文件和可执行文件，但保留由 CMake 生成的构建文件。

重新构建项目

如果你修改了源代码，可以在构建目录中运行：

`ninja` 或 `make` 或 `mingw32-make`

CMake 会自动检测更改并仅重新编译必要的文件。

其他 CMake 选项

- **构建类型**：你可以在生成构建文件时指定构建类型（如 Debug 或 Release）：

  ```
  cmake -DCMAKE_BUILD_TYPE=Debug ..
  ```

  或

  ```
  cmake -DCMAKE_BUILD_TYPE=Release ..
  ```

二进制发布版本

提供了适用于 **Windows-amd64** 和 **Linux-amd64** 的二进制发布版本。

这两个文件的功能在第一部分中已有说明，见上文。

请注意不要直接双击测试模块，而应在终端中运行并使用引号传入参数。

你可以自行重命名下载的文件。

Windows

编译器：gcc version 14.2.0（由 MSYS2 项目构建的 Rev3）
二进制文件：

- main.exe：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/uploads/2025/04/21/1809455497PiFKIRQO.exe
- test.exe：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/uploads/2025/04/21/1810126700ncKU3EXl.exe

Linux

编译器：gcc version 12.2.0（Debian 12.2.0-14）
二进制文件：

- main：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/main_elf
- test：
  https://zjucube.oss-cn-hangzhou.aliyuncs.com/test_elf

闪退?

你可能会发现程序在输入完数据后闪退，这是因为程序已经输出了结果并完成全部运行逻辑，此后它会自行退出。

为了避免看不到结果，你应该在终端中使用它，而不是直接双击打开它。如果你不理解 终端 这个词语，请复制这段文字，并向AI寻求帮助。