东北大学计算机科学与工程学院

数据结构课程设计报告

题目　　无向图的关节点问题

课题组长　时智贤

课题组成员　王韵然 杨宗昊

专业名称　　计算机科学与技术

班级　　计 2401 2402 2403

指导教师　　李发明

2025 年 12 月

课程设计任务书

题目：

无向图的关节点问题

问题描述：

对无向连通图，若删除某个结点使其成为非连通图，则称该结点为关节点。假设某一地区公路交通网，求解关节点。

设计要求：

设计求解无向连通图关节点的模拟程序。

(1) 采用邻接表或邻接矩阵存储结构。

(2) 可以随机、文件及人工输入数据。

(3) 采用深度优先遍历求解关节点。

(4) 实现关节点的查询和统计功能。

(5) 实现将关节点改造为非关节的功能。

(6) 其它完善性或扩展性功能。

指导教师签字：

年　　月　　日

课题程序设计分工

学号	姓名	程序设计函数原型、类	功能说明

20245131	时智贤	Graph* createRandom(int vertices, double density = 0.3);
		Graph* createFromFile(const string& filename);
		bool saveToFile(const Graph& graph, const string& filename);
		void update_graph_display();
		bool draw_callback(GtkWidget* widget, cairo_t* cr, gpointer data);
		void create_new_graph(GtkWidget* widget, gpointer data);
		void load_graph_from_file(GtkWidget* widget, gpointer data);
		void save_graph_to_file(GtkWidget* widget, gpointer data);
		void find_articulation_points(GtkWidget* widget, gpointer data);
		void count_articulation_points(GtkWidget* widget, gpointer data);
		void convert_articulation_point(GtkWidget* widget, gpointer data);
		void initialize_gui(int argc, char** argv);
			随机创建
			从文件读入
			写入文件
			更新图显示
			绘图
			创建新图
			创建新图
			从文件加载图
			保存图到文件
			查找关节点
			统计关节点
			改造关节点
			初始化 GUI

20245190	王韵然	class Graph {
		public:
		Graph(int vertices);
		void addEdge(int u, int v);
		void removeEdge(int u, int v);
		bool edgeExists(int u, int v) const;
		int getVertexCount() const;
		int getEdgeCount() const;
		const vector<vector<int>>& getAdjacencyList() const ;
		void printGraph() const;
		bool convertToNonArticulation(int vertex);
		};
			创建指定顶点数的图
			添加边
			删除边
			检查边是否存在
			获取顶点数
			获取边数
			获取邻接表
			打印图结构
			改造关节点为非关节点

20245248	杨宗昊	class Graph {
		private:
		void DFS(int u);
		public:
		void findArticulationPoints();
		vector<int> getArticulationPoints() const;
		int countArticulationPoints() const;
		bool isArticulationPoint(int vertex) const;
		};
			深度优先遍历
			查找并标记关节点
			获取关节点列表
			统计关节点数量
			判断是否为关节点

课题报告分工

章节	内容	完成人
1 课题概述	1.1 课题任务	王韵然 杨宗昊

	1.2 课题原理 1.3 相关知识	
2 需求分析	2.1 课题调研 2.2 用户需求分析	时智贤
3 方案设计	3.1 总体功能设计 3.2 数据结构设计 3.3 函数原型设计 3.4 输入输出设计 3.5 主算法设计 3.6 用户界面设计	杨宗昊
4 方案实现	4.1 开发环境与工具 4.2 程序设计关键技术 4.3 个人设计实现（按组员分工） 4.3.1 4.3.2 4.3.3	时智贤 王韵然 杨宗昊
5 测试与调试	5.1 个人测试（按组员分工） 5.1.1 5.1.2 5.1.3 5.2 组装与系统测试 5.3 系统运行	时智贤 王韵然 杨宗昊
6 课题总结	6.1 课题评价 6.2 团队协作 6.3 下一步工作 6.4 个人设计心得（按组员分工） 6.4.1 6.4.2 6.4.3	时智贤 王韵然 杨宗昊

目录

C.1 运行环境说明

1 课题概述

1.1 课 题 任 务

设计一个求解无向连通图关节点的模拟程序，要求实现图的存储、数据输入、深度优先遍历求解关节点、关节点的查询与统计、改造关节点为非关节点等功能，并提供图形化界面。

1.2 课 题 原 理

关节点定义：在无向连通图中，若删除某个顶点后图不再连通，则该顶点称为关节点（Articulation Point）。

Tarjan 算法：利用 DFS 遍历图，记录每个顶点的“发现时间”（disc）和“最小可达时间”（low），通过以下条件判断关节点：

1. 根节点：若有两个或以上子节点，则为关节点。
2. 非根节点：若其某个子节点的 `low[v] >= disc[u]`，则 `u` 为关节点。

数据结构：采用邻接表存储图，支持动态增删边。

GUI 实现：使用 GTK+库实现图形界面，支持图的绘制、交互与状态展示。

1.3 相 关 知 识

图论基础：无向图、连通性、邻接表。

深度优先搜索（DFS）及其时间戳应用。

C++面向对象编程与 STL 容器。

GTK+图形界面开发。

文件 I/O 与随机数生成。

2 需求分析

2.1 课 题 调 研

现有图论工具（如 Graphviz）功能强大但不支持关节点的动态改造。

教学工具需要直观展示关节点的查找与改造过程。

本系统旨在填补教学演示工具的空白，提供可视化交互。

2.2 用 户 需 求 分 析

用户角色	需求描述
学生	理解关节点概念，可视化查看关节点位置
教师	演示图算法，生成随机图或从文件加载
开发者	可扩展算法，支持更多图操作

3 方案设计

3.1 总 体 功 能 设 计

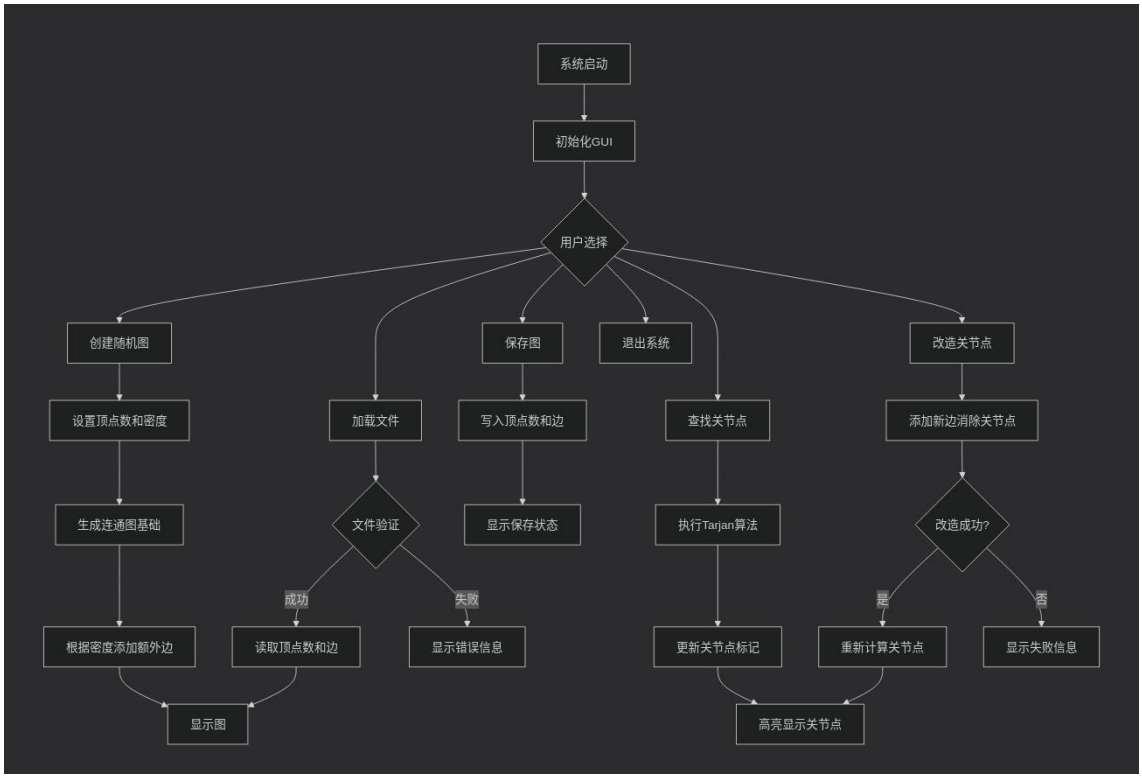
图管理：创建、加载、保存图。

关节点求解：DFS 查找关节点。

关节点统计：查询、计数、判断。

关节点改造：通过添加边消除关节点。

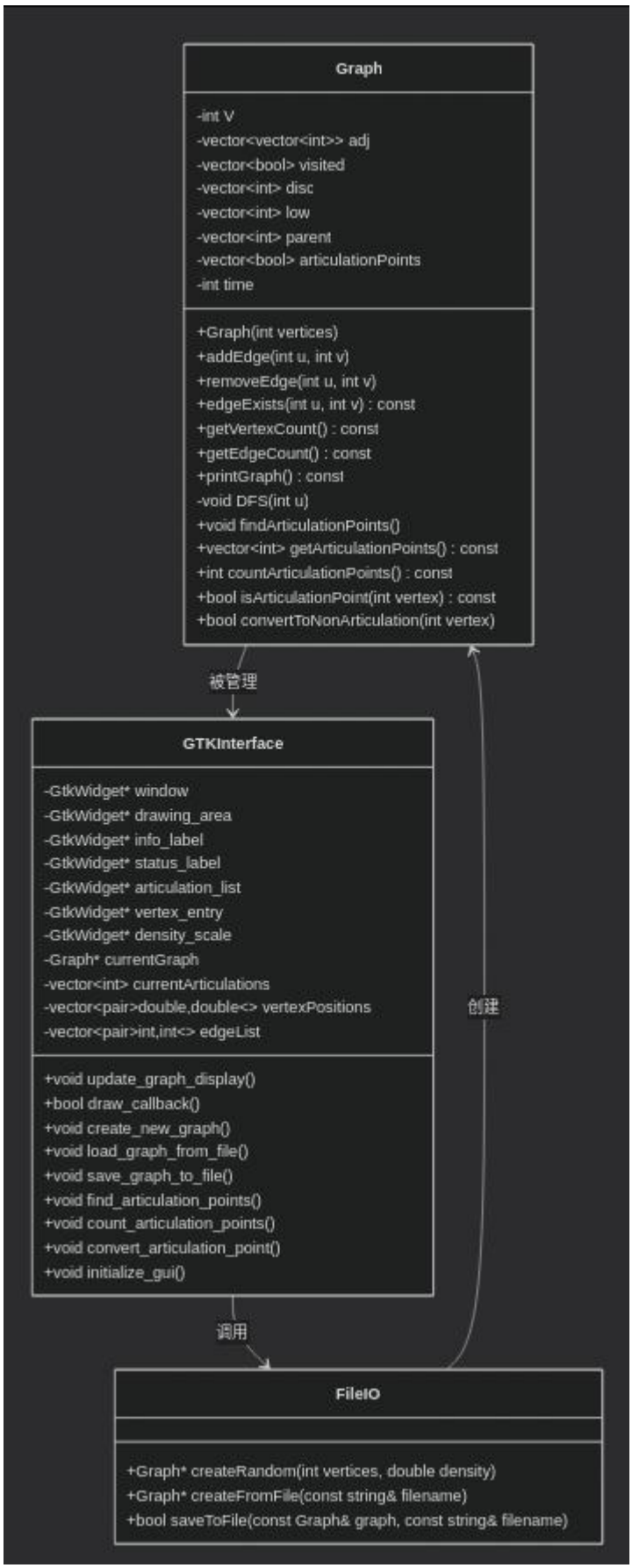
GUI 显示：绘制图结构，高亮关节点。



3.2 数据结构设计

cpp

```
class Graph {
    int V;                                  顶点数
    vector<vector<int>> adj;                邻接表
    vector<bool> visited;                   访问标记
    vector<int> disc, low, parent;          DFS 时间戳
    vector<bool> articulationPoints;        关节点标记
    int time;                               DFS 计时器
};
```


Graph
-int V
-vector<vector<int>> adj
-vector<bool> visited
-vector<int> disc
-vector<int> low
-vector<int> parent
-vector<bool> articulationPoints
-int time
+Graph(int vertices)
+addEdge(int u, int v)
+removeEdge(int u, int v)
+edgeExists(int u, int v) : const
+getVertexCount() : const
+getEdgeCount() : const
+printGraph() : const
-void DFS(int u)
+void findArticulationPoints()
+vector<int> getArticulationPoints() : const
+int countArticulationPoints() : const
+bool isArticulationPoint(int vertex) : const
+bool convertToNonArticulation(int vertex)
被管理
GTKInterface
-GtkWidget* window
-GtkWidget* drawing_area
-GtkWidget* info_label
-GtkWidget* status_label
-GtkWidget* articulation_list
-GtkWidget* vertex_entry
-GtkWidget* density_scale
-Graph* currentGraph
-vector<int> currentArticulations
-vector<pair>double,double<> vertexPositions
-vector<pair>int,int<> edgeList
+void update_graph_display()
+bool draw_callback()
+void create_new_graph()
+void load_graph_from_file()
+void save_graph_to_file()
+void find_articulation_points()
+void count_articulation_points()
+void convert_articulation_point()
+void initialize_gui()
创建
调用
FileIO
+Graph* createRandom(int vertices, double density)
+Graph* createFromFile(const string& filename)
+bool saveToFile(const Graph& graph, const string& filename)

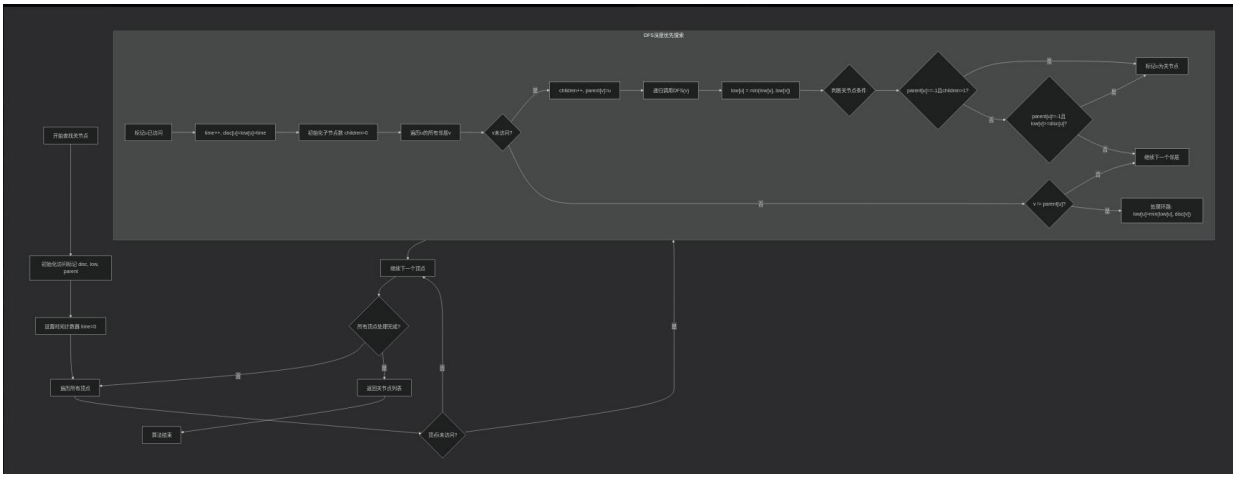
3.3 函数原型设计

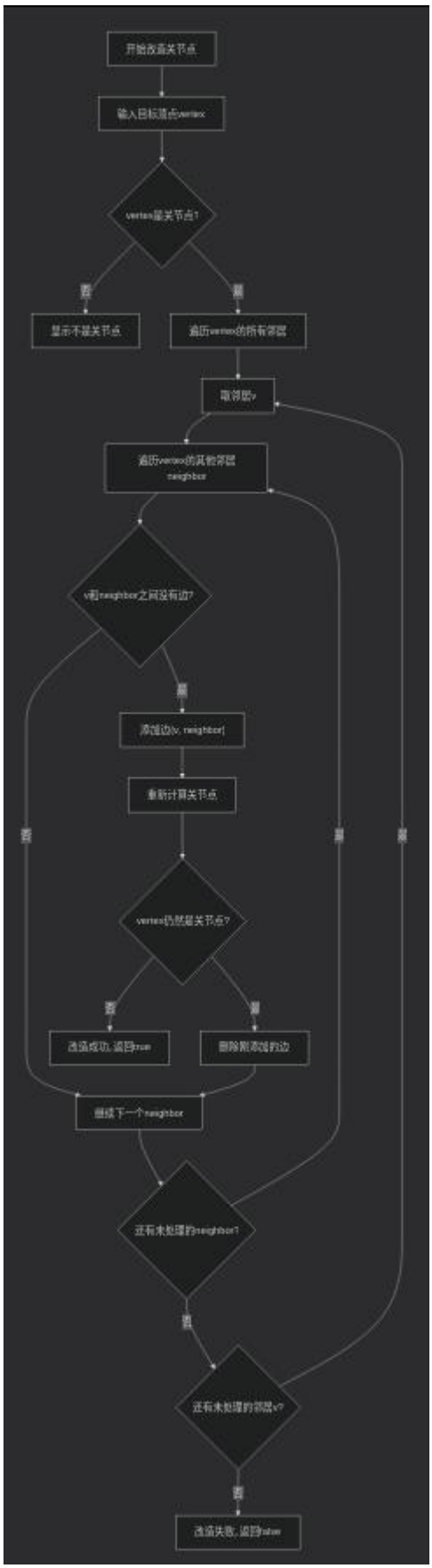
类别	函数名	功能
图操作	`addEdge` / `removeEdge`	增删边
图查询	`edgeExists` / `getEdgeCount`	检查边、统计边数
关节点	`findArticulationPoints`	查找所有关节点
关节点	`getArticulationPoints`	返回关节点列表
关节点	`countArticulationPoints`	统计关节点数
关节点	`isArticulationPoint`	判断是否为关节点
改造功能	`convertToNonArticulation`	改造关节点

3.4 主算法设计

Tarjan 算法：DFS 遍历中更新 `disc` 和 `low`，根据条件标记关节点。

改造算法：尝试在关节点的邻居之间添加边，使其不再成为关节点。




开始改造关节点
输入目标顶点vertex
vertex是关节点?
否
是
显示不是关节点
遍历vertex的所有邻居
取邻居v
遍历vertex的其他邻居
neighbor
v和neighbor之间没有边?
是
添加边(v, neighbor)
重新计算关节点
否
是
是
vertex仍然是关节点?
否
是
改造成功, 返回true
删除刚添加的边
继续下一个neighbor
还有未处理的neighbor?
否
还有未处理的邻居v?
否
改造失败, 返回false

3.5 用户界面设计

左侧：控制面板（按钮、列表、输入框）

右侧：图形化显示区域

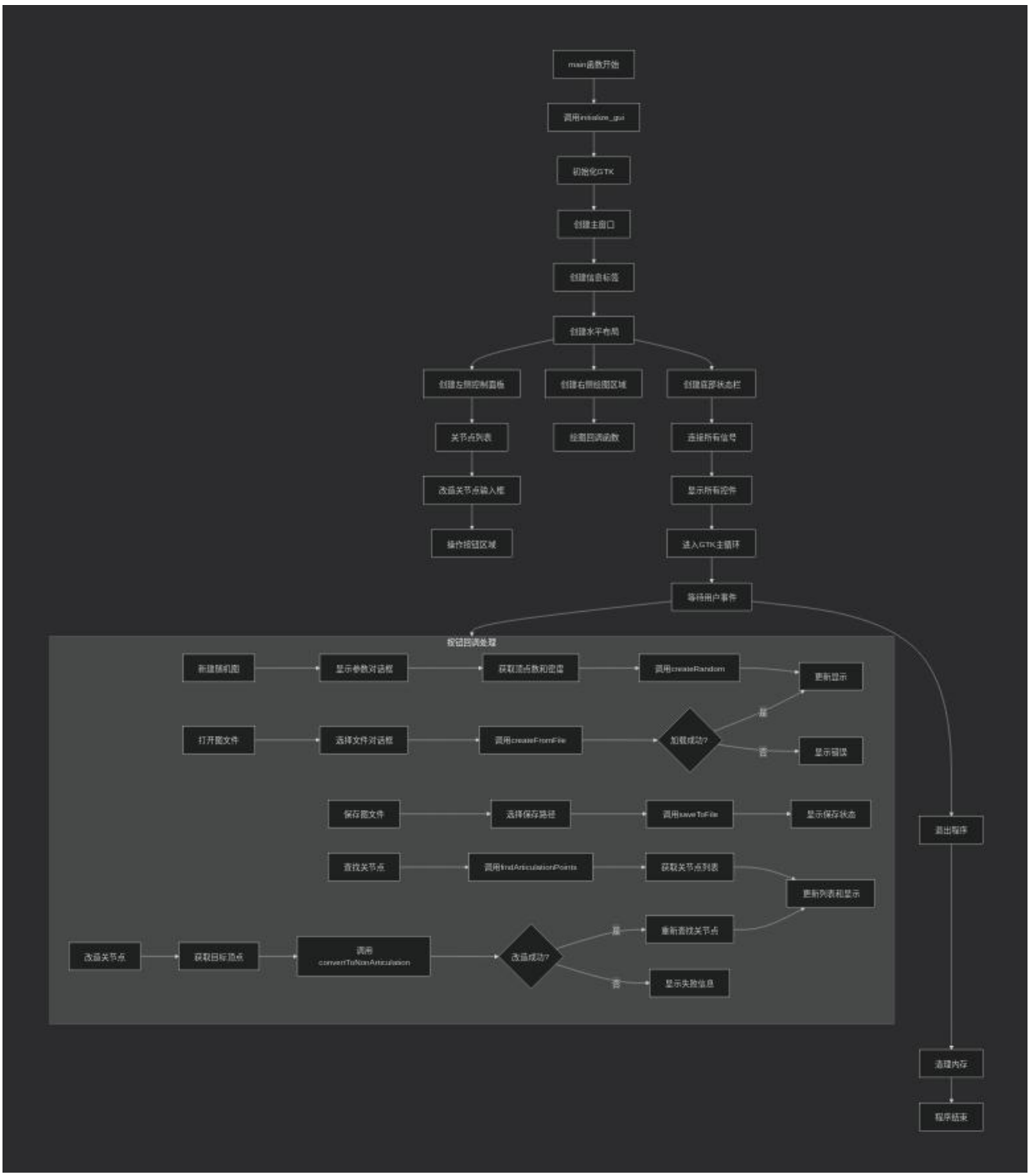
底部：状态栏

支持：新建、打开、保存、查找、统计、改造

3.6 输入输出设计

输入：随机生成、文件读取（TXT 格式）、GUI 交互

输出：文件保存、GUI 图形展示、控制台日志



4 方案实现

4.1 开发环境与工具

操作系统：Linux / Windows

编译器：g++ (C++11)

开发工具：VS Code

依赖库：GTK+3.0

4.2 程序设计关键技术

邻接表动态管理

DFS 递归实现

GTK+信号回调机制

文件流与错误处理

4.3 个人设计实现（按组员分工）

4.3.1 王韵然设计实现

实现关节点改造功能及相关基础图操作。在 Graph 类中，设计 addEdge、removeEdge、edgeExists 等方法用于边的管理，确保图的动态更新与一致性检查。同时，实现 convertToNonArticulation 方法，通过遍历目标关节点的邻接顶点，尝试添加额外边来破坏其关节点性质，使其不再成为连通图的关键节点。每次添加边后，会重新运行关节点检测算法以验证改造效果，确保功能正确性。

4.3.2 杨宗昊设计实现

实现基于深度优先遍历的关节点求解算法。在 Graph 类中，编写 DFS 方法，使用 disc 和 low 数组记录每个顶点的发现时间和最早可达祖先，结合 parent 和 children 数量判断关节点的两个条件：根节点有多个子节点、非根节点的子节点无法回到更早祖先。此外，实现 findArticulationPoints、getArticulationPoints、countArticulationPoints 和 isArticulationPoint 等方法，支持关节点的查找、列表获取、数量统计和状态查询。

4.3.3 时智贤设计实现

负责实现图的随机生成、文件读写与 GUI 界面。在 io.cpp 中，编写 createRandom 函数，根据顶点数和密度生成随机连通图；createFromFile 和 saveToFile 用于从文本文件读取图数据和保存图数据，支持格式校验与错误处理。在 main.cpp 中，设计基于 GTK 的图形界面，包括绘图区域、控制面板、状态栏等，实现了图的圆形布局显示、边与顶点的绘制、关节点的颜色区分，以及各功能按钮的信号连接与事件处理。

5 测试与调试

5.1 个人测试（按组员分工）

5.1.1 王韵然测试

编写了多个测试用例来验证基础操作和关节点改造功能。例如，创建不同结构的图，测试 addEdge 和 removeEdge 是否避免重复边，验证 edgeExists 返回正确。针对关节点改造，构造了多个包含关节点的图，手动或程序化调用 convertToNonArticulation，检查目标顶点是否成功变为非关节点，并在界面上更新显示。通过反复测试，确保改造逻辑在不同图结构下均能有效运行。

5.1.2 杨宗昊测试

设计了多组测试图，包括简单链式图、星型图、环状图等，验证 DFS 算法正确识别关节点。通过输出 disc 和 low 值，手动验证判断逻辑的准确性。同时，测试了 countArticulationPoints 返回的数量是否与预期一致，

isArticulationPoint 能否正确判断指定顶点是否为关节点。在 GUI 中，还测试了“查找关节点”和“统计关节点”按钮的功能与显示更新。

5.1.1 时智贤测试

测试随机生成图的功能，验证不同密度下生成的图是否连通且边数符合预期。文件读写方面，创建了多个测试文件，包含合法与非法数据，确保程序能正确读取、过滤错误并保存图结构。GUI 测试中，逐一验证了“新建随机图”“打开文件”“保存文件”“查找关节点”等按钮的功能是否正常，界面显示是否随图结构变化而实时更新，状态提示是否准确。

5.2 组装与系统测试

整合所有模块（算法 + GUI + IO）

测试文件读取异常处理（如 `sample-1.txt`）

测试顶点数限制（3-20）

5.3 系统运行

1. 编译命令：

bash

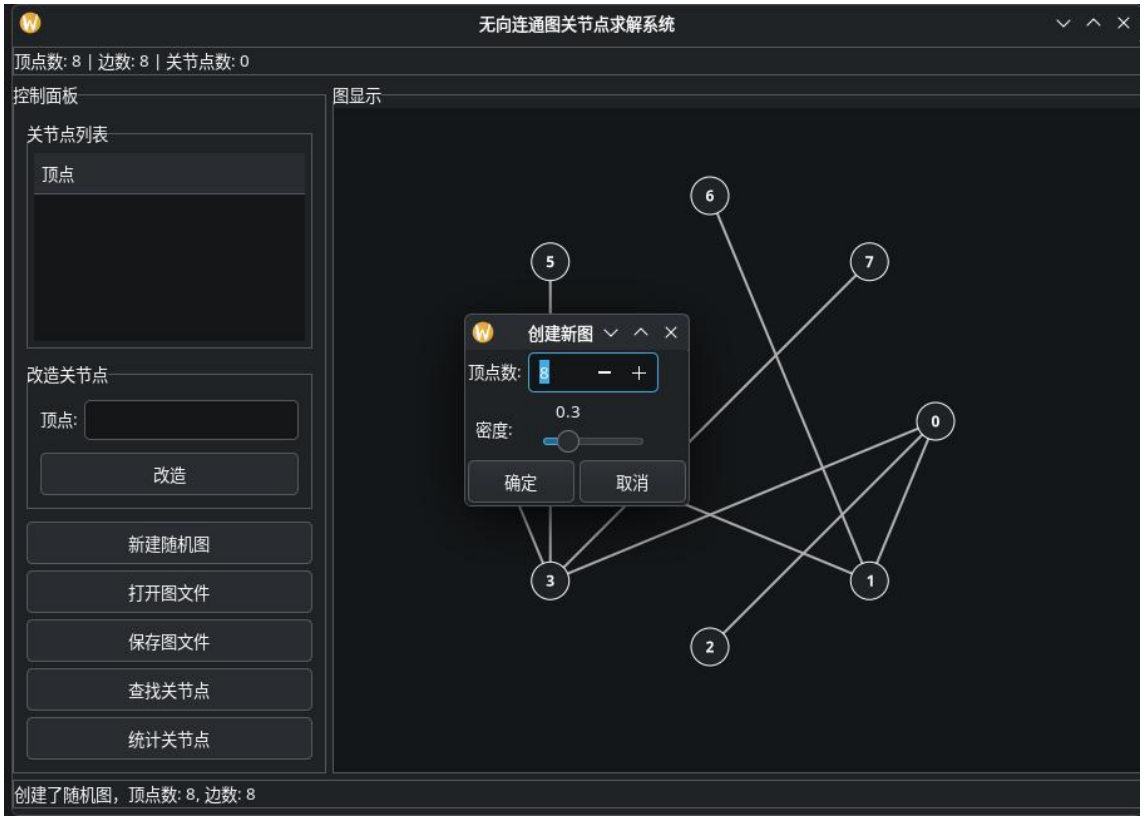
```
g++ *.cpp -o run.exe `pkg-config --cflags --libs gtk+-3.0`
```

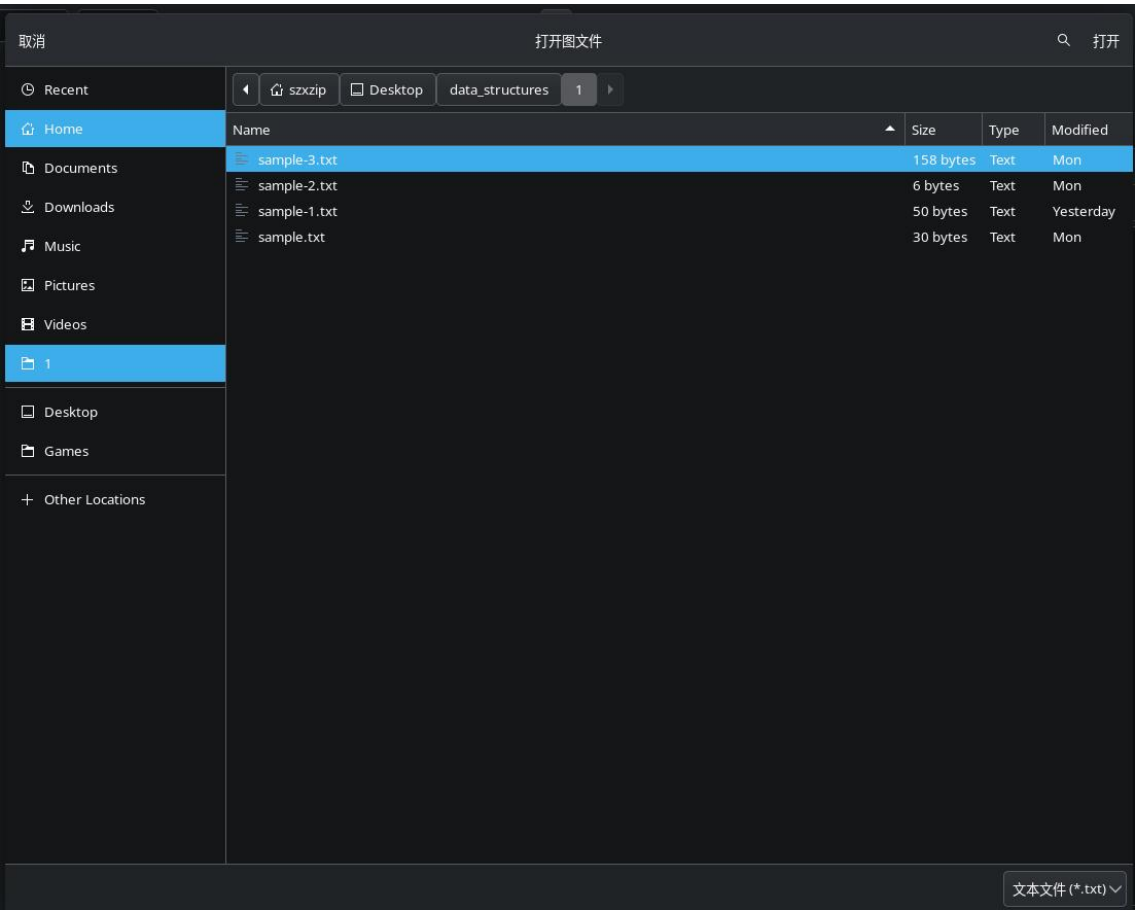
2. 运行界面：

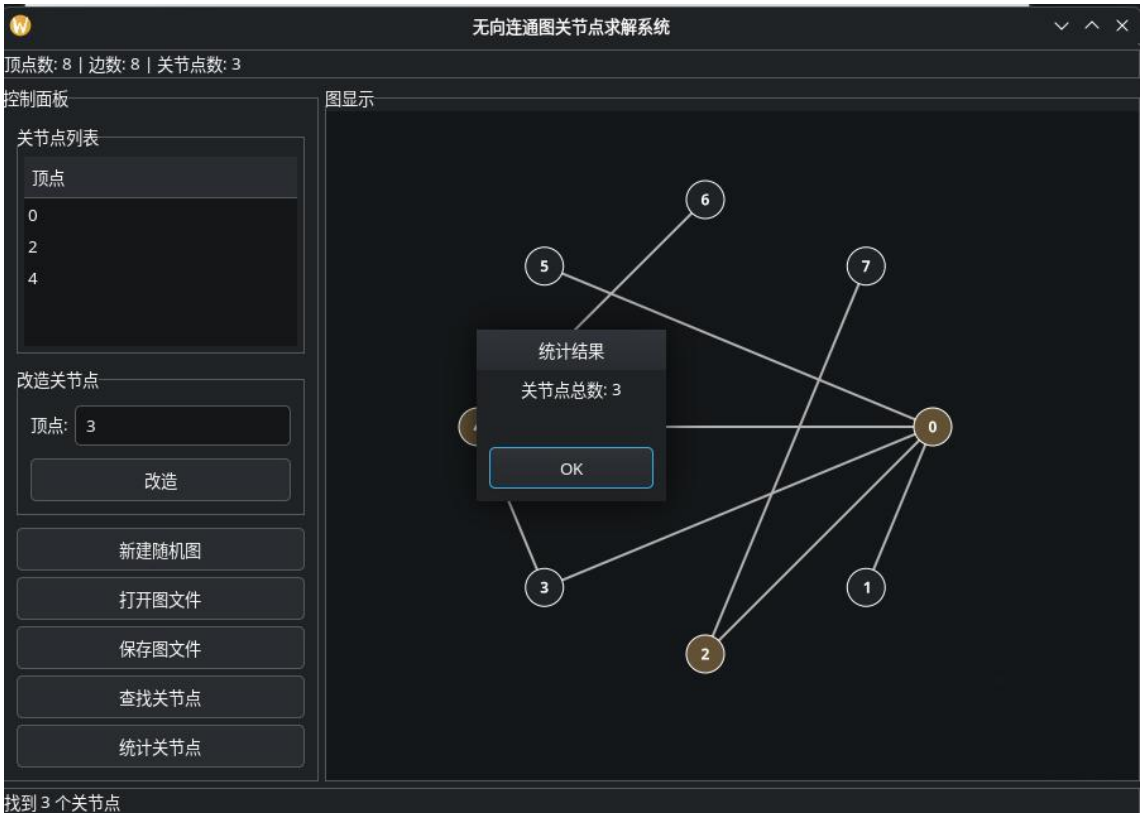
支持新建随机图、打开文件、保存文件

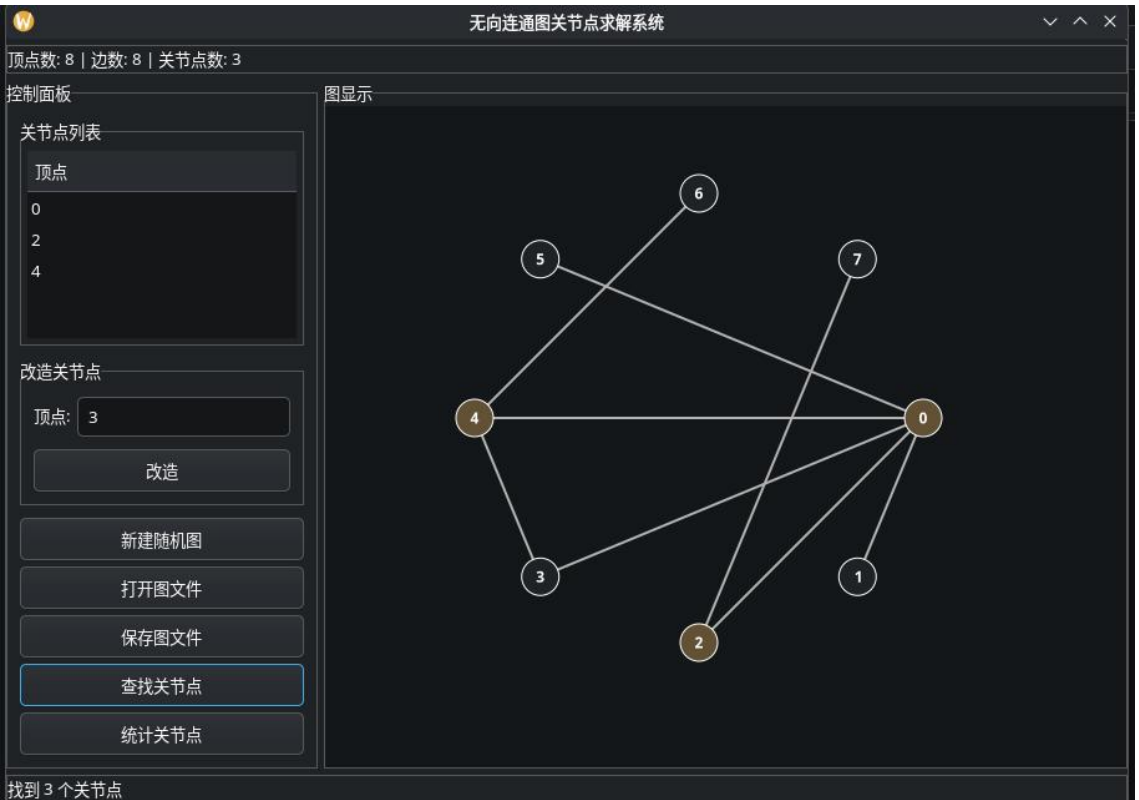
点击“查找关节点”可高亮显示

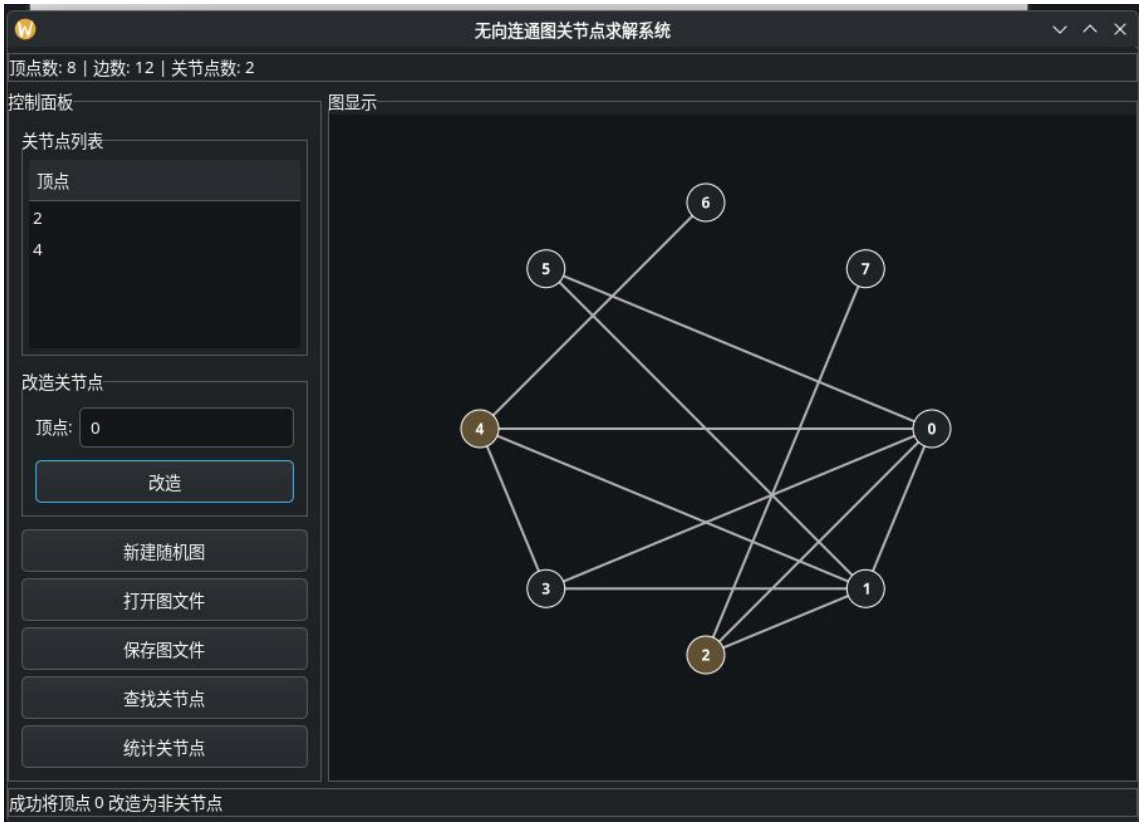
输入顶点编号可尝试改造关节点




取消
打开图文件
打开
Recent
Home
Documents
Downloads
Music
Pictures
Videos
1
Desktop
Games
Other Locations
szxzip
Desktop
data_structures
1
Name
Size
Type
Modified
sample-3.txt
158 bytes
Text
Mon
sample-2.txt
6 bytes
Text
Mon
sample-1.txt
50 bytes
Text
Yesterday
sample.txt
30 bytes
Text
Mon
文本文件 (*.txt)


无向连通图关节点求解系统
顶点数: 8 | 边数: 8 | 关节点数: 3
控制面板
关节点列表
顶点
0
2
4
改造关节点
顶点: 3
改造
新建随机图
打开图文件
保存图文件
查找关节点
统计关节点
图显示
统计结果
关节点总数: 3
OK
找到 3 个关节点


无向连通图关节点求解系统
顶点数: 8 | 边数: 8 | 关节点数: 3
控制面板
关节点列表
顶点
0
2
4
改造关节点
顶点: 3
改造
新建随机图
打开图文件
保存图文件
查找关节点
统计关节点
图显示
找到 3 个关节点



6 课题总结

6.1 课 题 评 价

优点:

算法正确性高，Tarjan 算法实现标准

GUI 界面友好，支持直观交互

功能完整，覆盖课题所有要求

不足:

顶点数限制较窄（3-20）

改造算法可能失败（取决于图结构）

6.2 团 队 协 作

分工明确：算法、GUI、IO、测试各司其职

使用 Git 进行版本控制

定期讨论进度与问题

6.3 个人设计小结（按组员分工）

6.3.1 王韵然设计小结

通过本次设计，深入理解了无向图中关节点的性质及其与连通性的关系。实现关节点改造功能认识到，通过增加冗余边可以增强图的鲁棒性，这在交通网络等实际场景中具有重要意义。也体会到在动态修改图结构时，必须同步更新相关状态（如重新计算关节点），才能保持数据一致性，这是系统设计中不可忽视的关键点。

6.3.2 杨宗昊设计小结

通过实现 Tarjan 算法求解关节点，加深了对深度优先遍历和回溯机制的理解。关节点的判断条件虽然简单，但在实现时需注意处理根节点和非根节点的不同逻辑，以及处理重边和自环的情况。此外，将算法封装成可复用的方法，并与 GUI 界面紧密结合，提升了系统的交互性和实用性，对算法工程化有了更深的体会。

6.3.3 时智贤设计小结

通过本次设计，掌握了图形界面开发与文件处理的基本方法。GUI 设计不仅要考虑功能完整性，还需注重用户体验，如布局合理、操作流畅、反馈明确。文件读写模块则强调了数据格式的规范性与错误处理的必要性。将算法、数据与界面三者有机结合，认识到一个完整系统需在功能、稳定性和易用性之间取得平衡，这也是软件开发中不可或缺的综合能力。

7 附录

B 课题设计文档

B-1 课程设计报告（电子版）

B-2 源程序代码（*.H，*.CPP）

header.h

```
#include <cmath>
#include <cstring>
#include <ctime>
#include <fstream>
#include <gtk/gtk.h>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

class Graph {
```

```
private:
    int V; //  顶点数
    vector<vector<int>> adj; //  邻接表
    vector<bool> visited; //  是否发现
    vector<int> disc; //  发现时间（时间戳）
    vector<int> low; //  最小非父可达节点
    vector<int> parent; //  父节点
    vector<bool> articulationPoints; //  关节点
    int time; // DFS 时间计数器

    // B
    void DFS(int u); //  深度优先遍历

public:
    // A
    Graph(int vertices); //  创建指定顶点数的图
    void addEdge(int u, int v); //  添加边
    void removeEdge(int u, int v); //  删除边
    bool edgeExists(int u, int v) const; //  检查边是否存在

    int getVertexCount() const { return V; } //  获取顶点数
    int getEdgeCount() const; //  获取边数
    const vector<vector<int>>& getAdjacencyList() const { return adj; } //  获取邻接表
    void printGraph() const; //  打印图结构

    // B
    void findArticulationPoints(); //  查找并标记关节点
    vector<int> getArticulationPoints() const; //  获取关节点列表
    int countArticulationPoints() const; //  统计关节点数量
    bool isArticulationPoint(int vertex) const; //  判断是否为关节点

    // A
    bool convertToNonArticulation(int vertex); //  改造关节点为非关节点
};

Graph* createRandom(int vertices, double density = 0.3); //  随机创建

/* *
 *  读写文件  (C)
 *
 * sample.txt:
 *  顶点数
 *  顶点  顶点  （即一条边）
 *  顶点  顶点
 *  ...
```

```
 */
Graph* createFromFile(const string& filename);
bool saveToFile(const Graph& graph, const string& filename);

// GUI
void update_graph_display(); // 更新图显示
bool draw_callback(GtkWidget* widget, cairo_t* cr, gpointer data); // 绘图
void create_new_graph(GtkWidget* widget, gpointer data); // 创建新图
void load_graph_from_file(GtkWidget* widget, gpointer data); // 从文件加载图
void save_graph_to_file(GtkWidget* widget, gpointer data); // 保存图到文件
void find_articulation_points(GtkWidget* widget, gpointer data); // 查找关节点
void count_articulation_points(GtkWidget* widget, gpointer data); // 统计关节点
void convert_articulation_point(GtkWidget* widget, gpointer data); // 改造关节点
void initialize_gui(int argc, char** argv); // 初始化 GUI

// 模板函数，可参考

template <typename T>
T my_min(T a, T b)
{
	return a < b ? a : b;
}

template <typename T>
int my_find(const vector<T>& vec, const T& value)
{
	for (size_t i = 0; i < vec.size(); i++) {
		if (vec[i] == value)
			return i;
	}
	return -1;
}
```

algorithm.cpp

```
#include "header.h"

void Graph::DFS(int u)
{
	visited[u] = true; // 把现在到达的节点设置为已经访问过
```

```
    time++; // 时间增加，用来判断经过的先后顺序
    disc[u] = low[u] = time; // disc 为本节点到达的时间点, low 为本节点能连通到的最早经过节点的经过时间，用以判断环路
    int children = 0; // 初始化，将当前节点的邻居（孩子）数量设置为 0

    for (int i = 0; i < adj[u].size(); i++) { // 深度优先遍历
        int v = adj[u][i];
        if (!visited[v]) { // 找到尚未经过的邻居节点
            children++; // 记录两个节点的关系
            parent[v] = u;
            DFS(v); // 对该邻居递归调用

            low[u] = my_min(low[u], low[v]); // 通过孩子可能到达更早经过的节点，因此更新值（my_min：自定义模板函数）

            if (parent[u] == -1 && children > 1) { // 对于根节点，用孩子的数量进行判断即可，>=2 为关节点
                articulationPoints[u] = true;
            }

            if (parent[u] != -1 && low[v] >= disc[u]) { // 对于非根节点，若其子节点无法回到比 u 更早经过的节点,代表没有环路，为关节点
                articulationPoints[u] = true;
            }
        } else if (v != parent[u]) { // 处理环路情况，即通过环路回到了更早已经经过的邻居节点的情况
            low[u] = my_min(low[u], disc[v]);
        }
    }
    cout << "深度优先搜索：完成！ " << endl;
}

void Graph::findArticulationPoints()
{ // 查找并标记关节点
    // 重置所有数组
    visited.assign(V, false);
    disc.assign(V, 0);
    low.assign(V, 0);
    parent.assign(V, -1);
    articulationPoints.assign(V, false);
    time = 0;

    // 对每个未访问的顶点调用 DFS，虽然有递归，但要防止图一开始就不联通的情况
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            DFS(i);
```

```
            }
        }
        cout << "查找并标记关节点：完成！ " << endl;
}

vector<int> Graph::getArticulationPoints() const // 获取关节点列表
{
        vector<int> result;
        for (int i = 0; i < V; i++) { // 拿 articulation 数组直接遍历
            if (articulationPoints[i]) {
                result.push_back(i);
            }
        }
        cout << "获取关节点列表：完成！ " << endl;
        return result;
}

int Graph::countArticulationPoints() const
{ // 统计关节点数量
        int count = 0;
        for (int i = 0; i < V; i++) { // 拿 articulation 数组直接遍历
            if (articulationPoints[i]) {
                count++;
            }
        }
        cout << "统计关节点数量： " << count << endl;
        return count;
}

bool Graph::isArticulationPoint(int vertex) const
{ // 判断是否为关节点
        if (vertex >= 0 && vertex < V) {
            cout << "判断是否为关节点：是。" << endl;
            return articulationPoints[vertex];
        }
        cout << "判断是否为关节点：否。" << endl;
        return false;
}
```

io.cpp

```
#include "header.h"
```

```
// 随机创建图
Graph* createRandom(int vertices, double density)
{
    Graph* graph = new Graph(vertices);

    // 生成连通图（最小密度）
    for (int i = 1; i < vertices; i++) { // 遍历顶点（除了首个顶点）
        int parent = rand() % i; // 每个顶点与后方顶点随机连一条线
        graph->addEdge(i, parent);
    }

    // 根据密度添加额外边
    int maxEdges = vertices * (vertices - 1) / 2; // 最大可能边数
    int targetEdges = static_cast<int>(density * maxEdges);
    int currentEdges = vertices - 1; // 最小可能边数，也是当前边数

    while (currentEdges < targetEdges) {
        int x = rand() % vertices;
        int y = rand() % vertices;

        if (x != y && !graph->edgeExists(x, y)) { // 避免自环、重复边
            graph->addEdge(x, y);
            currentEdges++;
        }
    }

    cout << "随机创建图：完成！ " << endl;
    return graph;
}

// 从文件读入图
Graph* createFromFile(const string& filename)
{
    ifstream file(filename);
    if (!file.is_open()) {
        cerr << "错误：无法打开文件 " << filename << endl; // 错误输出到命令行
        return nullptr; // 空指针
    }

    int V = 0; // 顶点数
    file >> V; // 若开头不合规，则之后的判定会拒绝通过

    // 顶点数 3-20
    if (V < 3 || V > 20) {
        cerr << "错误：无效的顶点数！ " << endl;
```

```
        file.close();
        return nullptr;
    }

    Graph* graph = new Graph(V);

    int lineNumber = 1; // 当前行号
    string line;
    getline(file, line); // 清除第一行换行符（和多余字符，如果有的话）

    // 读入边
    while (getline(file, line)) {
        lineNumber++;

        // 使用字符串流解析行
        istringstream iss(line);
        vector<int> values;
        int value;

        // 写入临时数组（仅能写入整数）
        while (iss >> value) {
            values.push_back(value);
        }

        // 检查每行元素数量，跳过多数/少数行、空行
        if (values.size() != 2) {
            cerr << "警告：第 " << lineNumber << " 行格式不正确！已跳过此行。" << endl;
            continue;
        }

        int x = values[0];
        int y = values[1];

        // 检查顶点编号是否有效
        if (x < 0 || x >= V || y < 0 || y >= V) {
            cerr << "警告：第 " << lineNumber << " 行包含非法顶点编号！已跳过此行。" << endl;
            continue;
        }

        if (x == y) {
            cerr << "警告：第 " << lineNumber << " 行是自环！已跳过此行。" << endl;
            continue;
        }

        graph->addEdge(x, y);
```

```
    }

    file.close();
    cout << "从文件读入图：完成！ " << endl;
    return graph;
}

// 写入图到文件
bool saveToFile(const Graph& graph, const string& filename)
{
    ofstream file(filename);
    if (!file.is_open()) {
        cerr << "错误：无法创建文件 " << filename << endl;
        return false;
    }

    // 写入顶点
    int V = graph.getVertexCount();
    file << V << endl;

    // 写入边
    for (int i = 0; i < V; i++) { // 遍历顶点
        const vector<int>& neighbors = graph.getAdjacencyList()[i]; // 读取该点邻接表
        for (size_t j = 0; j < neighbors.size(); j++) { // 遍历相邻顶点
            int neighbor = neighbors[j]; // 相邻顶点
            if (i < neighbor) { // 顶点小于相邻顶点，避免重复
                file << i << " " << neighbor << endl;
            }
        }
    }

    file.close();
    cout << "写入图到文件：完成！ " << endl;
    return true;
}
```

operation.cpp

```
#include "header.h"

// 构造函数：初始化一个具有指定顶点数的图
Graph::Graph(int vertices)
```

```
    : V(vertices) // 顶点数
    , time(0) // DFS 时间戳，用于追踪顶点访问顺序
{
    adj.resize(V); // 调整邻接表大小为顶点数
    visited.resize(V, false); // 初始化所有顶点为未访问
    disc.resize(V, -1); // 初始化所有顶点的发现时间为 -1（未访问）
    low.resize(V, -1); // 初始化所有顶点的 low 值为 -1
    parent.resize(V, -1); // 初始化所有顶点的父节点为 -1（无父节点）
    articulationPoints.resize(V, false); // 初始化所有顶点为非关节点
    cout << "Graph 构造函数：完成！" << endl;
}

// 添加边
void Graph::addEdge(int u, int v)
{
    if (u >= 0 && u < V && v >= 0 && v < V && u != v) {
        // 如果边不存在，才进行添加，避免重复边
        if (!edgeExists(u, v)) {
            adj[u].push_back(v);
            adj[v].push_back(u);
        }
    }
    cout << "添加边：完成！" << endl;
}

// 删除边
void Graph::removeEdge(int u, int v)
{
    if (u >= 0 && u < V && v >= 0 && v < V) {
        // 从 u 的邻接表中删除 v
        int pos = my_find(adj[u], v); // 查找 v 在 u 的邻接表中的位置（my_find：自定义模板函数）
        if (pos != -1) { // 如果找到
            // 将最后一个元素移到当前位置，然后弹出最后一个元素
            adj[u][pos] = adj[u].back();
            adj[u].pop_back();
        }

        // 从 v 的邻接表中删除 u
        pos = my_find(adj[v], u);
        if (pos != -1) {
            adj[v][pos] = adj[v].back();
            adj[v].pop_back();
        }
    }
    cout << "删除边：完成！" << endl;
```

```
}

// 检查边是否存在
bool Graph::edgeExists(int u, int v) const
{
    if (u < 0 || u >= V || v < 0 || v >= V) {
        cout << "检查边是否存在：不存在" << endl;
        return false; // 无效顶点，边不存在
    }
    // 在 u 的邻接表中查找 v，如果找到则边存在
    cout << "检查边是否存在：存在" << endl;
    return my_find(adj[u], v) != -1;
}

// 获取边数
int Graph::getEdgeCount() const
{
    int count = 0;
    for (int i = 0; i < V; i++) {
        count += adj[i].size(); // 累加每个顶点的邻接表大小
    }
    cout << "获取边数：" << count / 2 << endl;
    return count / 2; // 无向图每条边在邻接表中出现两次
}

// 打印图结构
void Graph::printGraph() const
{
    for (int i = 0; i < V; i++) {
        cout << "顶点 " << i << ": ";
        for (size_t j = 0; j < adj[i].size(); j++) {
            cout << adj[i][j] << " ";
        }
    }
    cout << "打印图结构：完成！" << endl;
}

// 改造关节点为非关节点：尝试通过添加额外边将指定顶点变为非关节点
bool Graph::convertToNonArticulation(int vertex)
{
    // 如果该顶点本身不是关节点，则无需改造
    if (!isArticulationPoint(vertex)) {
        cout << "改造关节点 " << vertex << " ：失败！" << endl;
        return false;
    }
```

```
    // 遍历该顶点的所有邻接顶点 v
    for (size_t i = 0; i < adj[vertex].size(); i++) {
        int v = adj[vertex][i];
        // 对于每个邻接顶点 v，尝试将其连接到 vertex 的另一个邻居 neighbor
        for (size_t j = 0; j < adj[vertex].size(); j++) {
            int neighbor = adj[vertex][j];
            // 如果 neighbor 不是 v 且 v 和 neighbor 之间没有边，则添加新边
            if (neighbor != v && !edgeExists(v, neighbor)) {
                addEdge(v, neighbor);
                // 重新计算图中的所有关节点
                findArticulationPoints();
                // 检查 vertex 是否仍然是关节点
                if (!isArticulationPoint(vertex)) {
                    cout << "改造关节点 " << vertex << " ： 成功！ " << endl;
                    return true;
                }
            }
        }
    }
    cout << "改造关节点 " << vertex << " ： 失败！ " << endl;
    return false;
}
```

main.cpp

```
#include "header.h"

// 全局变量
GtkWidget* window = nullptr;
GtkWidget* drawing_area = nullptr;
GtkWidget* info_label = nullptr;
GtkWidget* status_label = nullptr;
GtkWidget* articulation_list = nullptr;
GtkWidget* vertex_entry = nullptr;
GtkWidget* density_scale = nullptr;

Graph* currentGraph = nullptr; // 当前图
vector<int> currentArticulations; // 当前关节点
vector<pair<double, double>> vertexPositions; // GUI 顶点位置
vector<pair<int, int>> edgeList; // 当前边列表
```

```
// 更新图显示
void update_graph_display()
{
    if (!currentGraph)
        return;

    // 计算顶点位置（圆形布局）
    vertexPositions.clear(); // 清空顶点位置
    int V = currentGraph->getVertexCount(); // 顶点数
    int centerX = 300, centerY = 250, radius = 180; // 圆心及半径

    for (int i = 0; i < V; i++) {
        double angle = 2 * 3.141592653589793 * i / V; // 等分圆周
        double x = centerX + radius * cos(angle);
        double y = centerY + radius * sin(angle);
        vertexPositions.push_back(make_pair(x, y)); // 构造后写入
    }

    // 获取边列表
    edgeList.clear(); // 清空边列表
    const vector<vector<int>>& adj = currentGraph->getAdjacencyList();
    for (int i = 0; i < V; i++) { // 遍历顶点
        for (size_t j = 0; j < adj[i].size(); j++) { // 遍历相邻顶点
            int neighbor = adj[i][j]; // 相邻顶点
            if (i < neighbor) { // 顶点小于相邻顶点，避免重复
                edgeList.push_back(make_pair(i, neighbor)); // 写入边
            }
        }
    }

    // 更新信息标签
    stringstream info;
    info << "顶点数: " << V << " | 边数: " << currentGraph->getEdgeCount() << " | 关节点数: " <<
currentArticulations.size();

    gtk_label_set_text(GTK_LABEL(info_label), info.str().c_str());

    // 重绘图
    gtk_widget_queue_draw(drawing_area);
}

// 绘图
bool draw_callback(GtkWidget* widget, cairo_t* cr, gpointer data)
{
    // 清空背景
```

```
cairo_set_source_rgb(cr, 0.08, 0.09, 0.1);
cairo_paint(cr);

if (!currentGraph)
    return FALSE;

// 绘制边
cairo_set_source_rgb(cr, 0.7, 0.7, 0.7);
cairo_set_line_width(cr, 2);

for (size_t i = 0; i < edgeList.size(); i++) {
    int u = edgeList[i].first;
    int v = edgeList[i].second;

    double x1 = vertexPositions[u].first;
    double y1 = vertexPositions[u].second;
    double x2 = vertexPositions[v].first;
    double y2 = vertexPositions[v].second;

    cairo_move_to(cr, x1, y1);
    cairo_line_to(cr, x2, y2);
    cairo_stroke(cr);
}

// 绘制顶点
for (int i = 0; i < currentGraph->getVertexCount(); i++) {
    double x = vertexPositions[i].first;
    double y = vertexPositions[i].second;

    // 判断是否为关节点
    bool isAP = false;
    for (size_t j = 0; j < currentArticulations.size(); j++) {
        if (currentArticulations[j] == i) {
            isAP = true;
            break;
        }
    }

    if (isAP) {
        // 关节点
        cairo_set_source_rgb(cr, 0.39, 0.32, 0.21);
    } else {
        // 普通顶点
        cairo_set_source_rgb(cr, 0.12, 0.14, 0.15);
    }
```

```
            cairo_arc(cr, x, y, 15, 0, 2 * 3.141592653589793);
            cairo_fill(cr);

            // 顶点边框
            cairo_set_source_rgb(cr, 1, 1, 1);
            cairo_set_line_width(cr, 1);
            cairo_arc(cr, x, y, 15, 0, 2 * 3.141592653589793);
            cairo_stroke(cr);

            // 顶点标签
            cairo_set_source_rgb(cr, 1, 1, 1);
            cairo_select_font_face(cr, "Sans", CAIRO_FONT_SLANT_NORMAL,
CAIRO_FONT_WEIGHT_BOLD);
            cairo_set_font_size(cr, 12);

            string label = to_string(i);
            cairo_text_extents_t extents;
            cairo_text_extents(cr, label.c_str(), &extents);
            cairo_move_to(cr, x - extents.width / 2, y + extents.height / 2);
            cairo_show_text(cr, label.c_str());
        }

        return FALSE;
}

// 创建新图
void create_new_graph(GtkWidget* widget, gpointer data)
{
        GtkWidget* dialog = gtk_dialog_new_with_buttons("创建新图",
            GTK_WINDOW(window),
            GTK_DIALOG_MODAL,
            "确定", GTK_RESPONSE_OK,
            "取消", GTK_RESPONSE_CANCEL,
            NULL);

        GtkWidget* content = gtk_dialog_get_content_area(GTK_DIALOG(dialog));
        GtkWidget* grid = gtk_grid_new();
        gtk_grid_set_row_spacing(GTK_GRID(grid), 5);
        gtk_grid_set_column_spacing(GTK_GRID(grid), 5);

        // 顶点数输入
        GtkWidget* vertex_label = gtk_label_new("顶点数:");
        GtkWidget* vertex_spin = gtk_spin_button_new_with_range(3, 20, 1);
        gtk_spin_button_set_value(GTK_SPIN_BUTTON(vertex_spin), 8);
```

```
    // 密度选择
    GtkWidget* density_label = gtk_label_new("密度:");
    density_scale = gtk_scale_new_with_range(GTK_ORIENTATION_HORIZONTAL, 0.1, 0.9, 0.1);
    gtk_range_set_value(GTK_RANGE(density_scale), 0.3);

    gtk_grid_attach(GTK_GRID(grid), vertex_label, 0, 0, 1, 1);
    gtk_grid_attach(GTK_GRID(grid), vertex_spin, 1, 0, 1, 1);
    gtk_grid_attach(GTK_GRID(grid), density_label, 0, 1, 1, 1);
    gtk_grid_attach(GTK_GRID(grid), density_scale, 1, 1, 1, 1);

    gtk_container_add(GTK_CONTAINER(content), grid);
    gtk_widget_show_all(dialog);

    if (gtk_dialog_run(GTK_DIALOG(dialog)) == GTK_RESPONSE_OK) {
        int vertices = gtk_spin_button_get_value_as_int(GTK_SPIN_BUTTON(vertex_spin));
        double density = gtk_range_get_value(GTK_RANGE(density_scale));

        // 删除旧图
        if (currentGraph) {
            delete currentGraph;
        }

        currentGraph = createRandom(vertices, density);
        currentArticulations.clear();

        // 更新显示
        update_graph_display();

        // 清空关节点列表
        GtkListStore* store = GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(articulation_list)));
        gtk_list_store_clear(store);

        stringstream msg;
        msg << "创建了随机图，顶点数: " << vertices << "，边数: " << currentGraph->getEdgeCount();
        gtk_label_set_text(GTK_LABEL(status_label), msg.str().c_str());
    }

    gtk_widget_destroy(dialog);
}

// 从文件加载图
void load_graph_from_file(GtkWidget* widget, gpointer data)
{
```

```
    GtkWidget* dialog = gtk_file_chooser_dialog_new("打开图文件",
        GTK_WINDOW(window),
        GTK_FILE_CHOOSER_ACTION_OPEN,
        "取消", GTK_RESPONSE_CANCEL,
        "打开", GTK_RESPONSE_ACCEPT,
        NULL);

    GtkFileFilter* filter = gtk_file_filter_new();
    gtk_file_filter_set_name(filter, "文本文件  (*.txt)");
    gtk_file_filter_add_pattern(filter, "*.txt");
    gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(dialog), filter);

    if (gtk_dialog_run(GTK_DIALOG(dialog)) == GTK_RESPONSE_ACCEPT) {
        char* filename = gtk_file_chooser_get_filename(GTK_FILE_CHOOSER(dialog));

        // 删除旧图
        if (currentGraph) {
            delete currentGraph;
        }

        currentGraph = createFromFile(filename);
        if (currentGraph) {
            currentArticulations.clear();
            update_graph_display();

            // 清空关节点列表
            GtkListStore* store =
GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(articulation_list)));
            gtk_list_store_clear(store);

            stringstream msg;
            msg << "从文件加载图成功，顶点数: " << currentGraph->getVertexCount()
                << "，边数: " << currentGraph->getEdgeCount();
            gtk_label_set_text(GTK_LABEL(status_label), msg.str().c_str());
        } else {
            gtk_label_set_text(GTK_LABEL(status_label), "加载文件失败！");
        }

        g_free(filename);
    }

    gtk_widget_destroy(dialog);
}

// 保存图到文件
```

```
void save_graph_to_file(GtkWidget* widget, gpointer data)
{
    if (!currentGraph) {
        gtk_label_set_text(GTK_LABEL(status_label), "没有图可保存!  ");
        return;
    }

    GtkWidget* dialog = gtk_file_chooser_dialog_new("保存图文件",
        GTK_WINDOW(window),
        GTK_FILE_CHOOSER_ACTION_SAVE,
        "取消", GTK_RESPONSE_CANCEL,
        "保存", GTK_RESPONSE_ACCEPT,
        NULL);

    gtk_file_chooser_set_current_name(GTK_FILE_CHOOSER(dialog), "graph.txt");

    GtkFileFilter* filter = gtk_file_filter_new();
    gtk_file_filter_set_name(filter, "文本文件  (*.txt)");
    gtk_file_filter_add_pattern(filter, "*.txt");
    gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(dialog), filter);

    if (gtk_dialog_run(GTK_DIALOG(dialog)) == GTK_RESPONSE_ACCEPT) {
        char* filename = gtk_file_chooser_get_filename(GTK_FILE_CHOOSER(dialog));

        if (saveToFile(*currentGraph, filename)) {
            gtk_label_set_text(GTK_LABEL(status_label), "图保存成功!  ");
        } else {
            gtk_label_set_text(GTK_LABEL(status_label), "保存文件失败!  ");
        }

        g_free(filename);
    }

    gtk_widget_destroy(dialog);
}

// 查找关节点
void find_articulation_points(GtkWidget* widget, gpointer data)
{
    if (!currentGraph) {
        gtk_label_set_text(GTK_LABEL(status_label), "请先创建或加载图!  ");
        return;
    }

    currentGraph->findArticulationPoints();
```

```
    currentArticulations = currentGraph->getArticulationPoints();

    // 更新显示
    update_graph_display();

    // 更新关节点列表

gtk_list_store_clear(GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(articulation_list))));

    GtkListStore* store = GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(articulation_list)));
    for (size_t i = 0; i < currentArticulations.size(); i++) {
        GtkTreeIter iter;
        gtk_list_store_append(store, &iter);
        string vertex_str = to_string(currentArticulations[i]);
        gtk_list_store_set(store, &iter, 0, vertex_str.c_str(), -1);
    }

    stringstream msg;
    msg << "找到 " << currentArticulations.size() << " 个关节点";
    gtk_label_set_text(GTK_LABEL(status_label), msg.str().c_str());
}

// 统计关节点
void count_articulation_points(GtkWidget* widget, gpointer data)
{
    if (!currentGraph) {
        gtk_label_set_text(GTK_LABEL(status_label), "请先创建或加载图! ");
        return;
    }

    int count = currentGraph->countArticulationPoints();
    stringstream msg;
    msg << "关节点总数: " << count;

    // 显示消息对话框
    GtkWidget* dialog = gtk_message_dialog_new(GTK_WINDOW(window),
        GTK_DIALOG_MODAL,
        GTK_MESSAGE_INFO,
        GTK_BUTTONS_OK,
        "%s", msg.str().c_str());
    gtk_window_set_title(GTK_WINDOW(dialog), "统计结果");
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}
```

```
// 改造关节点
void convert_articulation_point(GtkWidget* widget, gpointer data)
{
    if (!currentGraph) {
        gtk_label_set_text(GTK_LABEL(status_label), "请先创建或加载图！ ");
        return;
    }

    const char* vertex_text = gtk_entry_get_text(GTK_ENTRY(vertex_entry));
    if (strlen(vertex_text) == 0) {
        gtk_label_set_text(GTK_LABEL(status_label), "请输入顶点编号！ ");
        return;
    }

    int vertex = stoi(vertex_text);

    if (vertex < 0 || vertex >= currentGraph->getVertexCount()) {
        gtk_label_set_text(GTK_LABEL(status_label), "顶点编号无效！ ");
        return;
    }

    if (!currentGraph->isArticulationPoint(vertex)) {
        stringstream msg;
        msg << "顶点 " << vertex << " 不是关节点！ ";
        gtk_label_set_text(GTK_LABEL(status_label), msg.str().c_str());
        return;
    }

    if (currentGraph->convertToNonArticulation(vertex)) {
        // 重新查找关节点
        currentGraph->findArticulationPoints();
        currentArticulations = currentGraph->getArticulationPoints();

        // 更新显示
        update_graph_display();

        // 更新关节点列表

gtk_list_store_clear(GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(articulation_list))));

        GtkListStore* store =
GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(articulation_list)));
        for (size_t i = 0; i < currentArticulations.size(); i++) {
            GtkTreeIter iter;
```

```
                gtk_list_store_append(store, &iter);
                string vertex_str = to_string(currentArticulations[i]);
                gtk_list_store_set(store, &iter, 0, vertex_str.c_str(), -1);
            }

            stringstream msg;
            msg << "成功将顶点 " << vertex << " 改造为非关节点";
            gtk_label_set_text(GTK_LABEL(status_label), msg.str().c_str());
        } else {
            stringstream msg;
            msg << "改造顶点 " << vertex << " 失败";
            gtk_label_set_text(GTK_LABEL(status_label), msg.str().c_str());
        }
}

// 初始化 GUI
void initialize_gui(int argc, char** argv)
{
    gtk_init(&argc, &argv);

    // 创建主窗口
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "无向连通图关节点求解系统");
    gtk_window_set_default_size(GTK_WINDOW(window), 900, 600);
    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

    // 创建主垂直盒子
    GtkWidget* main_vbox = gtk_box_new(GTK_ORIENTATION_VERTICAL, 0);
    gtk_container_add(GTK_CONTAINER(window), main_vbox);

    // 信息标签
    info_label = gtk_label_new("欢迎使用关节点求解系统");
    gtk_label_set_xalign(GTK_LABEL(info_label), 0);
    GtkWidget* info_frame = gtk_frame_new(NULL);
    gtk_container_add(GTK_CONTAINER(info_frame), info_label);
    gtk_box_pack_start(GTK_BOX(main_vbox), info_frame, FALSE, FALSE, 5);

    // 创建水平盒子，左侧是控制面板，右侧是绘图区域
    GtkWidget* hbox = gtk_box_new(GTK_ORIENTATION_HORIZONTAL, 5);
    gtk_box_pack_start(GTK_BOX(main_vbox), hbox, TRUE, TRUE, 0);

    // 左侧控制面板
    GtkWidget* control_frame = gtk_frame_new("控制面板");
    gtk_widget_set_size_request(control_frame, 250, -1);
```

```
    GtkWidget* control_vbox = gtk_box_new(GTK_ORIENTATION_VERTICAL, 10);
    gtk_container_set_border_width(GTK_CONTAINER(control_vbox), 10);
    gtk_container_add(GTK_CONTAINER(control_frame), control_vbox);

    // 关节点列表
    GtkWidget* list_frame = gtk_frame_new("关节点列表");
    GtkWidget* list_vbox = gtk_box_new(GTK_ORIENTATION_VERTICAL, 5);
    gtk_container_set_border_width(GTK_CONTAINER(list_vbox), 5);

    // 创建列表
    GtkListStore* store = gtk_list_store_new(1, G_TYPE_STRING);
    articulation_list = gtk_tree_view_new_with_model(GTK_TREE_MODEL(store));

    GtkCellRenderer* renderer = gtk_cell_renderer_text_new();
    GtkTreeViewColumn* column = gtk_tree_view_column_new_with_attributes("顶点", renderer, "text", 0,
NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW(articulation_list), column);

    GtkWidget* scrolled_window = gtk_scrolled_window_new(NULL, NULL);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scrolled_window),
        GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
    gtk_scrolled_window_set_min_content_height(GTK_SCROLLED_WINDOW(scrolled_window), 150);
    gtk_container_add(GTK_CONTAINER(scrolled_window), articulation_list);

    gtk_box_pack_start(GTK_BOX(list_vbox), scrolled_window, TRUE, TRUE, 0);
    gtk_container_add(GTK_CONTAINER(list_frame), list_vbox);
    gtk_box_pack_start(GTK_BOX(control_vbox), list_frame, TRUE, TRUE, 0);

    // 改造关节点区域
    GtkWidget* convert_frame = gtk_frame_new("改造关节点");
    GtkWidget* convert_vbox = gtk_box_new(GTK_ORIENTATION_VERTICAL, 10);
    gtk_container_set_border_width(GTK_CONTAINER(convert_vbox), 10);

    GtkWidget* convert_hbox = gtk_box_new(GTK_ORIENTATION_HORIZONTAL, 5);
    GtkWidget* vertex_label = gtk_label_new("顶点:");
    vertex_entry = gtk_entry_new();
    gtk_entry_set_max_length(GTK_ENTRY(vertex_entry), 3);
    gtk_entry_set_width_chars(GTK_ENTRY(vertex_entry), 5);

    gtk_box_pack_start(GTK_BOX(convert_hbox), vertex_label, FALSE, FALSE, 0);
    gtk_box_pack_start(GTK_BOX(convert_hbox), vertex_entry, TRUE, TRUE, 0);

    GtkWidget* convert_button = gtk_button_new_with_label("改造");

    gtk_box_pack_start(GTK_BOX(convert_vbox), convert_hbox, FALSE, FALSE, 0);
```

```
gtk_box_pack_start(GTK_BOX(convert_vbox), convert_button, FALSE, FALSE, 0);

gtk_container_add(GTK_CONTAINER(convert_frame), convert_vbox);
gtk_box_pack_start(GTK_BOX(control_vbox), convert_frame, FALSE, FALSE, 0);

// 操作按钮区域
GtkWidget* button_vbox = gtk_box_new(GTK_ORIENTATION_VERTICAL, 5);

GtkWidget* new_button = gtk_button_new_with_label("新建随机图");
GtkWidget* open_button = gtk_button_new_with_label("打开图文件");
GtkWidget* save_button = gtk_button_new_with_label("保存图文件");
GtkWidget* find_button = gtk_button_new_with_label("查找关节点");
GtkWidget* count_button = gtk_button_new_with_label("统计关节点");

gtk_box_pack_start(GTK_BOX(button_vbox), new_button, FALSE, FALSE, 0);
gtk_box_pack_start(GTK_BOX(button_vbox), open_button, FALSE, FALSE, 0);
gtk_box_pack_start(GTK_BOX(button_vbox), save_button, FALSE, FALSE, 0);
gtk_box_pack_start(GTK_BOX(button_vbox), find_button, FALSE, FALSE, 0);
gtk_box_pack_start(GTK_BOX(button_vbox), count_button, FALSE, FALSE, 0);

gtk_box_pack_start(GTK_BOX(control_vbox), button_vbox, FALSE, FALSE, 0);

gtk_box_pack_start(GTK_BOX(hbox), control_frame, FALSE, FALSE, 0);

// 右侧绘图区域
GtkWidget* drawing_frame = gtk_frame_new("图显示");
drawing_area = gtk_drawing_area_new();
gtk_widget_set_size_request(drawing_area, 600, 500);
gtk_container_add(GTK_CONTAINER(drawing_frame), drawing_area);
gtk_box_pack_start(GTK_BOX(hbox), drawing_frame, TRUE, TRUE, 0);

// 状态栏
status_label = gtk_label_new("就绪");
gtk_label_set_xalign(GTK_LABEL(status_label), 0);
GtkWidget* status_frame = gtk_frame_new(NULL);
gtk_container_add(GTK_CONTAINER(status_frame), status_label);
gtk_box_pack_start(GTK_BOX(main_vbox), status_frame, FALSE, FALSE, 5);

// 连接信号
g_signal_connect(new_button, "clicked", G_CALLBACK(create_new_graph), NULL);
g_signal_connect(open_button, "clicked", G_CALLBACK(load_graph_from_file), NULL);
g_signal_connect(save_button, "clicked", G_CALLBACK(save_graph_to_file), NULL);
g_signal_connect(find_button, "clicked", G_CALLBACK(find_articulation_points), NULL);
g_signal_connect(count_button, "clicked", G_CALLBACK(count_articulation_points), NULL);
g_signal_connect(convert_button, "clicked", G_CALLBACK(convert_articulation_point), NULL);
```

```
    g_signal_connect(drawing_area, "draw", G_CALLBACK(draw_callback), NULL);

    // 显示所有控件
    gtk_widget_show_all(window);
}

// 主函数
int main(int argc, char** argv)
{
    initialize_gui(argc, argv);
    gtk_main();

    // 清理内存
    if (currentGraph) {
        delete currentGraph;
    }

    return 0;
}
```

sample.txt

```
7
0 1
0 2
1 2
1 3
1 4
3 4
4 5
```

C.1 运行环境说明

部署 Windows GUI，以及编译

> 提示：GUI 采用 GTK3，可参考下面步骤来配置环境。

1. 安装 [msys2](https://www.msys2.org/)。（用来创建一个 linux 环境）

2. 打开 MSYS2 **UCRT64**。（有好几个环境，ucrt64 更常用些）

> 以下都是在此 ucrt64 环境中执行

3. 安装 gtk3。

```
pacman -S mingw-w64-ucrt-x86_64-gtk3
```

4. 安装 C++ 编译工具。

```
pacman -S mingw-w64-ucrt-x86_64-toolchain base-devel
```

5. 切换到项目的目录。

```
cd X:/path/to/codes
```

`cd` = change directory。`X:` 是盘符，比如 C 盘。`/` 用来代替 `\`（linux 和 win 的路径分隔符不一样）。

比如

```
cd C:/users/me/Desktop/data_structures/1/
```

6. 编译。

```
g++ -o run.exe sample1.cpp sample2.cpp `pkg-config --cflags --libs gtk+-3.0` -mwindows
```

`g++` 是 C++ 编译工具。`-o run.exe` 输出可执行文件。`sample1.cpp` 编译 sample1.cpp。`pkg-config --cflags --libs gtk+-3.0` 添加 gtk3。