同济大学

TONGJI UNIVERSITY

数据结构课程设计

项目名称 关键活动

学　　院 计算机科学与技术学院

专　　业 软件工程

学生姓名 杨瑞晨

学　　号 2351050

指导教师 张颖

日　　期 2024 年 12 月 6 日

目 录

1 项目分析 …… 1
1.1 项目背景分析 …… 1
1.2 项目功能分析 …… 1
1.2.1 功能要求 …… 1
1.2.2 输入要求 …… 1
1.2.3 输出要求 …… 1
1.2.4 项目实例 …… 2
2 项目设计 …… 3
2.1 数据结构设计 …… 3
2.1.1 队列（Queue） …… 3
2.1.2 图（Graph） …… 3
2.2 类设计 …… 3
2.2.1 Edge 结构体 …… 3
2.2.2 Queue 模板类 …… 4
2.2.3 Graph 类 …… 5
3 项目实现 …… 8
3.1 流程表示 …… 8
3.2 功能实现 …… 8
3.2.1 拓扑排序 …… 8
3.2.2 计算时间 …… 10
3.2.3 关键任务 …… 10
3.3 main 函数 …… 11
4 项目测试 …… 13
4.1 正常测试 …… 13
4.1.1 简单情况测试 …… 13
4.1.2 一般情况测试，单个起点和单个终点 …… 13
4.1.3 不可行的方案测试 …… 15
4.2 健壮性测试 …… 15
5 项目心得与体会 …… 16

1 项目分析

1.1 项目背景分析

在项目管理和任务调度中，确定项目的最短完成时间和识别关键路径是至关重要的。本项目旨在实现一个任务调度与关键路径分析系统，该系统能够通过拓扑排序和最早/最晚开始时间计算来确定项目的最短完成时间和关键活动。

1.2 项目功能分析

1.2.1 功能要求

本实验项目是要求在任务调度问题中，如果还给出了完成每个字任务需要的时间，则可以算出完成整个工程项目需要的最短时间。在这些子任务中，有些任务即使推迟几天完成，也不会影响全局的工期；但是有些任务必须准时完成，否则整个项目的工期就要因此而延误，这些任务叫做“关键活动”。

请编写程序判定一个给定的工程项目的任务调度是否可行；如果该调度方案可行，则计算完成整个项目需要的最短时间，并且输出所有的关键活动。

1.2.2 输入要求

输入第 1 行给出两个正整数 N（N≤100）和 M，其中 N 是任务交接点（即衔接两个项目依赖的两个子任务的结点，例如：若任务 2 要在任务 1 完成后才开始，则两个任务之间必有一个交接点）的数量，交接点按 1 ~ N 编号，M 是字任务的数量，依次编号为 1 ~ M。随后 M 行，每行给出 3 个正整数，分别是该任务开始和完成设计的交接点编号以及完成该任务所需要的时间，整数间用空格分隔。

1.2.3 输出要求

如果任务调度不可行，则输出 0；否则第一行输出完成整个项目所需要的时间，第 2 行开始输出所有关键活动，每个关键活动占一行，按照格式“v->W”输出，其中 V 和 W 为该任务开始和完成涉及的交接点编号。关键活动输出的顺序规则是：任务开始的交接点编号小者优先，起点编号相同时，与输入时任务的顺序相反。如下面测试用例 2 中，任务 <5，7> 先于任务 <5，8> 输入，而作为关键活动输出时则次序相反。

1.2.4 项目实例

序号	输入	输出	说明
1	7 8 1 2 4 1 3 3 2 4 5 3 4 3 4 5 2 4 6 6 5 7 5 6 7 2	17 1 –>2 2 –>4 4 –>6 6 –>7	简单情况测试
2	9 11 1 2 6 1 3 4 1 4 5 2 5 1 3 5 1 4 6 2 5 7 9 5 8 7 6 8 4 7 9 2 8 9 4	18 1 –>2 2 –>5 5 –>8 5 –>7 7 –>9 8 –>9	一般情况测试，单个起点和单个终点
3	4 5 1 2 4 2 3 5 3 4 6 4 2 3 4 1 2	0	不可行的方案测试

2 项目设计

2.1 数据结构设计

根据对题目的分析，在本项目中，我们采用了两种核心的数据结构：队列（Queue）和图（Graph）。

2.1.1 队列（Queue）

拓扑排序需要一个队列来存储所有入度为零的节点，这些节点表示没有前置任务或所有前置任务已完成的任务。队列需要还需要动态扩容以适应不确定数量的任务交接点。队列操作需要高效，以保证拓扑排序的性能。

每个队列都是一个动态数组，支持动态扩容，使用模板支持任意数据类型。队列是一种先进先出（FIFO, First-In-First-Out）的数据结构，它允许在一端（队尾）插入元素，而在另一端（队头）移除元素，支持入队、出队、查看队头元素等操作。

队列具有以下一系列优点：

- 有序性：队列中的元素按照它们被添加的顺序排列，先进入的元素先被移除。
- 动态性：大多数队列实现都是动态的，可以根据需要自动调整大小。
- 限制性访问：队列只允许在两端进行操作，队头用于移除元素，队尾用于添加元素。

2.1.2 图（Graph）

图是表示任务调度的核心数据结构，需要存储所有的任务交接点和子任务。为了计算最早和最晚开始时间，图需要能够快速地更新和查询任务的入度以及相邻任务之间的依赖关系。图还需要能够执行拓扑排序，并存储排序结果。

图类具有以下一系列优点：

- 图类整合了任务交接点和子任务的管理，提供了一个统一的视图来处理任务调度问题。
- 实现了拓扑排序和时间计算的功能，可以直接用于关键路径分析。
- 通过邻接矩阵和入度数组，图类可以快速地更新任务状态，并计算任务的最早和最晚开始时间。

2.2 类设计

2.2.1 Edge 结构体

在任务调度图中，每个子任务可以看作是一条边，连接两个任务交接点。边需要存储从哪个交接点开始，到哪个交接点结束，以及完成这个子任务所需的时间。为了后续计算最早和最晚开始时间，边结构需要能够快速地被访问和更新。

```cpp
// 边结构体，表示一个子任务
struct Edge
{
    int from, to, weight; // 子任务的开始交接点、结束交接点和所需时间
    int index;            // 子任务的编号
};
```

2.2.2 Queue 模板类

Queue 类提供了一个灵活且动态的队列实现，支持快速的插入和删除操作。在拓扑排序中，它用于存储所有入度为零的节点，以便按顺序处理。Queue 类提供一系列操作：

- 构造函数：创建一个队列对象，可以指定队列的初始容量。
- 析构函数：销毁队列对象，释放内存。
- push: 将元素添加到队列尾部，并在必要时扩容。
- pop: 移除并返回队列头部的元素。
- front: 返回队列头部的元素，不移除。
- empty: 检查队列是否为空。
- size: 返回队列中元素的数量。
- clear: 清空队列。
- resize: 扩展队列容量。

```cpp
const int DEFAULT_CAPACITY = 10;

// 自定义队列类，用于拓扑排序中的队列操作
template <typename T>
class Queue
{
private:
    T *data;        // 队列元素数组
    int frontIndex; // 队头索引
    int rearIndex;  // 队尾索引
    int capacity;   // 队列容量

    void resize(int newCapacity)
    {
        // 重新分配更大的数组
        T *newData = new T[newCapacity];
        for (int i = 0; i < size(); ++i)
        {
            newData[i] = data[(frontIndex + i) % capacity];
        }
        delete[] data;
        data = newData;
        frontIndex = 0;
        rearIndex = size();
        capacity = newCapacity;
    }

```

```
public:
    // 构造函数
    Queue(int initialCapacity = DEFAULT_CAPACITY) : frontIndex(0), rearIndex(0),
    ↪  capacity(initialCapacity)
    {
        data = new T[capacity];
    }
    // 析构函数
    ~Queue() { delete[] data; }
    void push(const T &value) // 入队
    {
        // 如果队满，扩容
        if (size() == capacity - 1) resize(2 * capacity);
        data[rearIndex] = value; // 入队
        rearIndex = (rearIndex + 1) % capacity;
    }
    T pop() // 出队
    {
        // 如果队空，抛出异常
        if (empty()) throw std::out_of_range("Queue is empty");
        T value = data[frontIndex]; // 出队
        frontIndex = (frontIndex + 1) % capacity;
        return value;
    }
    T &front() // 获取队头元素
    {
        // 如果队空，抛出异常
        if (empty()) throw std::out_of_range("Queue is empty");
        return data[frontIndex];
    }
    bool empty() const // 判断队是否为空
    {
        return frontIndex == rearIndex;
    }
    bool full() const // 判断队是否为满
    {
        return (rearIndex + 1) % capacity == frontIndex;
    }
    int size() const // 获取队大小
    {
        return (rearIndex - frontIndex + capacity) % capacity;
    }
    void clear() // 清空队
    {
        frontIndex = rearIndex = 0;
    }
};
```

2.2.3 Graph 类

Graph 类是系统的核心，它不仅存储了任务调度图的所有必要信息，还提供了关键路径分析的核心功能。通过拓扑排序和时间计算，它可以确定项目的最短完成时间和关键活动。

Graph 类实现的功能：

（1）addTask(int index, int start, int end, int time): 添加子任务信息，并更新邻接矩阵和入度数组。

（2）topologicalSort(): 执行拓扑排序，并返回是否成功（无环）。

（3）calculateTimes(): 计算最早开始时间和最晚开始时间。

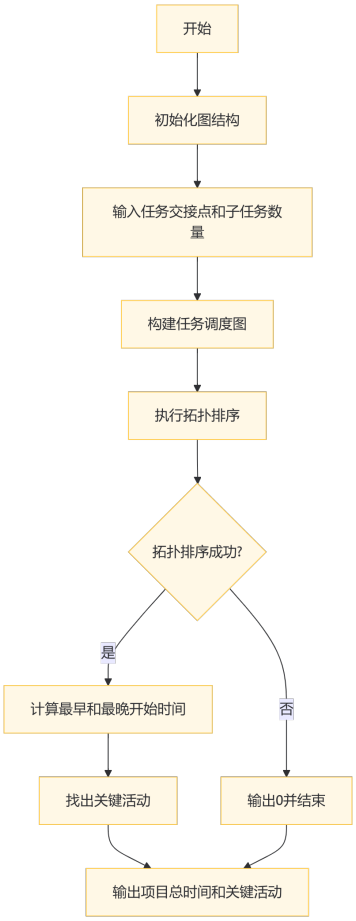
（4）findCriticalActivities(): 找出所有关键活动并输出。

```
// 图类，用于表示任务调度图
class Graph
{
private:
    int N, M;       // 任务交接点数量和子任务数量
    Edge *tasks;    // 子任务数组，保存每个子任务的开始交接点、结束交接点和所需时间
    int **adj;      // 邻接矩阵，表示任务之间的依赖关系和时间
    int *inDegree; // 入度数组，表示每个任务的前置任务数量

    int *earliest; // 最早开始时间数组
    int *latest;   // 最晚开始时间数组

    int *topoOrder; // 拓扑排序结果数组

public:
    // 构造函数，初始化图
    Graph(int n, int m) : N(n), M(m)
    {
        tasks = new Edge[M];     // 创建子任务数组
        adj = new int *[N + 1]; // 创建邻接矩阵
        for (int i = 1; i <= N; ++i)
        {
            adj[i] = new int[N + 1];
            for (int j = 1; j <= N; ++j)
            {
                adj[i][j] = INF; // 初始化邻接矩阵，表示没有连接
            }
        }
        inDegree = new int[N + 1];          // 创建入度数组
        earliest = new int[N + 1];          // 创建最早开始时间数组
        latest = new int[N + 1];            // 创建最晚开始时间数组
        fill(latest, latest + N + 1, INF); // 初始化最晚开始时间数组为无穷大
        topoOrder = new int[N];             // 创建拓扑排序结果数组
    }

    // 析构函数，释放动态内存
    ~Graph()
    {
        delete[] tasks;
        for (int i = 1; i <= N; ++i)
        {
            delete[] adj[i];
        }
        delete[] adj;
        delete[] inDegree;
        delete[] earliest;
        delete[] latest;
        delete[] topoOrder;
    }

    // 添加子任务信息，并更新邻接矩阵和入度数组
    void addTask(int index, int start, int end, int time)
    {
```

```
54          tasks[index] = {start, end, time, index}; // 将子任务信息存储到任务数组中
55          adj[start][end] = time;                   // 更新邻接矩阵，表示任务 start 到任务 end 所
            ↪ 需时间为 time
56          inDegree[end]++;                          // 更新入度数组，任务 end 的入度加 1，表示有一
            ↪ 个任务依赖于它
57      }
58
59      // 拓扑排序函数，返回拓扑排序是否成功
60      bool topologicalSort();
61
62      // 计算最早开始时间和最晚开始时间
63      void calculateTimes();
64
65      // 找出所有关键活动并输出
66      void findCriticalActivities();
67  };
```

3 项目实现

3.1 流程表示



3.2 功能实现

3.2.1 拓扑排序

拓扑排序是针对有向无环图（DAG）的顶点进行排序的一种算法，使得每个顶点的邻接点都出现在它的前面。这种排序不是唯一的，但一定存在。在任务调度中，拓扑排序可以帮助我们确定任务的执行顺序，使得所有前置任务都在实际执行的任务之前完成。

Kahn 算法（使用队列实现的拓扑排序）：

- 初始化：创建一个队列 Q，用于存储所有入度为 0 的顶点。
- 入度统计：遍历图中的每个顶点，对于每个顶点，检查它的入度。如果入度为 0，则将其加入队列 Q。
- 节点处理：当队列 Q 非空时，执行以下步骤：

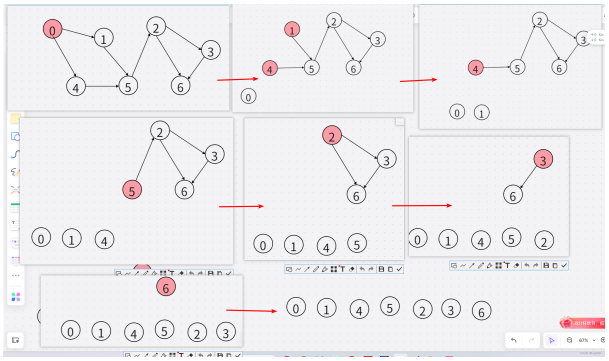
（1）从队列 Q 中移除一个顶点 u。

（2）将顶点 u 加入到拓扑排序的结果中。

（3）遍历顶点 u 的所有邻接点 v，对每个邻接点 v，将其入度减 1。如果邻接点 v 的入度变为 0，则将其加入队列 Q。

• 环检测：如果在所有顶点都被处理后，拓扑排序的结果中包含的顶点数小于图中顶点的总数，说明图中存在环，即存在依赖循环。

以下是其步骤的示意图：



代码实现：

```
// 拓扑排序函数，返回拓扑排序是否成功
bool topologicalSort()
{
    Queue<int> q;
    // 将所有入度为 0 的节点入队
    for (int i = 1; i <= N; ++i)
    {
        if (inDegree[i] == 0)
        {
            q.push(i);
        }
    }

    int index = 0; // 用于记录拓扑排序结果的下标
    // 遍历队列，执行拓扑排序
    while (!q.empty())
    {
        int u = q.front();      // 获取队列头部元素
        q.pop();                // 弹出队列头部元素
        topoOrder[index++] = u; // 将节点 u 加入拓扑排序结果

        // 遍历所有与节点 u 相邻的节点，更新它们的入度
        for (int v = 1; v <= N; ++v)
        {
            if (adj[u][v] != INF)
            { // 如果存在边 u -> v
                if (--inDegree[v] == 0)
                {
                    q.push(v); // 将入度变为 0 的节点入队
                }
            }
```

```
            }
        }

        // 如果拓扑排序结果包含所有节点，则返回 true；否则返回 false（图中有环，无法拓扑排序）
        return index == N;
    }
```

3.2.2 计算时间

计算每个任务的最早开始时间和最晚开始时间。根据拓扑排序的结果，从起始节点到所有其他节点正向计算最早开始时间。然后，从结束节点到所有其他节点反向计算最晚开始时间。

```
    // 计算最早开始时间和最晚开始时间
    void calculateTimes()
    {
        // 计算最早开始时间（正向遍历拓扑排序结果）
        for (int i = 0; i < N; ++i)
        {
            int u = topoOrder[i];
            // 遍历所有与 u 相邻的节点，更新最早开始时间
            for (int v = 1; v <= N; ++v)
            {
                if (adj[u][v] != INF)
                { // 如果存在边 u -> v
                    earliest[v] = max(earliest[v], earliest[u] + adj[u][v]);
                }
            }
        }

        // 计算项目总时间（即最早开始时间的最大值）
        int projectTime = *max_element(earliest + 1, earliest + N + 1);
        fill(latest, latest + N + 1, projectTime); // 将所有节点的最晚开始时间初始化为项目总时间

        // 计算最晚开始时间（反向遍历拓扑排序结果）
        for (int i = N - 1; i >= 0; --i)
        {
            int u = topoOrder[i];
            // 遍历所有与 u 相邻的节点，更新最晚开始时间
            for (int v = 1; v <= N; ++v)
            {
                if (adj[u][v] != INF)
                { // 如果存在边 u -> v
                    latest[u] = min(latest[u], latest[v] - adj[u][v]);
                }
            }
        }
    }
```

3.2.3 关键任务

识别项目中的关键活动，即那些影响项目总时间的活动。遍历所有子任务，对于每个子任务，检查其结束节点的最晚开始时间是否等于其开始节点的最早开始时间加上该任务的持续时间。如果

相等，则该任务是关键活动，将其输出。

```
    // 找出所有关键活动并输出
    void findCriticalActivities()
    {
        Edge criticalActivities[MAX_M]; // 存储关键活动
        int criticalCount = 0;          // 关键活动计数

        // 遍历所有子任务，判断是否是关键活动
        for (int i = 0; i < M; ++i)
        {
            int u = tasks[i].from, v = tasks[i].to, w = tasks[i].weight;
            // 判断是否是关键活动：最早开始时间 + 任务时间 = 最晚开始时间
            if (w == latest[v] - earliest[u])
            {
                criticalActivities[criticalCount++] = tasks[i];
            }
        }

        // 输出项目总时间（即最早开始时间的最大值）
        int projectTime = *max_element(earliest + 1, earliest + N + 1);
        cout << " 项目总时间: " << projectTime << endl;

        // 对关键活动按照开始节点编号升序，起始节点相同的按照输入顺序降序排序
        sort(criticalActivities, criticalActivities + criticalCount, [](const Edge &a, const
        ↪  Edge &b)
             {
            if (a.from != b.from) return a.from < b.from;
            return a.index > b.index; });

        cout << " 关键活动: " << endl;
        // 输出所有关键活动
        for (int i = 0; i < criticalCount; ++i)
        {
            cout << criticalActivities[i].from << " -> " << criticalActivities[i].to << endl;
        }
    }
```

主函数是程序的入口点，负责协调程序的执行流程。通过标准输入接收 N 和 M，然后创建 Graph 对象，添加任务，执行拓扑排序，计算时间，最后输出关键活动和项目总时间。

3.3 main 函数

```
// 主函数
int main()
{
    int N, M;
    cout << " 请输入任务交接点数量和子任务数量: ";
    while (true)
    {
        cin >> N >> M; // 输入任务交接点数量 N 和子任务数量 M
        if (cin.fail() || N <= 0 || M <= 0)
        {
            cerr << " 输入无效，请重新输入: " << endl;
```

```
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }
        else
        {
            break;
        }
    }

    Graph graph(N, M); // 创建图对象

    // 输入所有子任务的起点、终点和所需时间，并添加到图中
    cout << " 请依次输入任务的开始和完成的交接点编号以及所需时间: " << endl;
    for (int i = 0; i < M; ++i)
    {
        int start, end, time;

        cin >> start >> end >> time;
        if (cin.fail() || start <= 0 || start > N || end <= 0 || end > N || time <= 0)
        {
            cerr << " 输入无效，请重新输入第 " << i + 1 << " 个子任务: " << endl;
            cin.clear();
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
            --i;
            continue;
        }
        else graph.addTask(i, start, end, time);
    }

    // 进行拓扑排序，如果图中有环，则输出 0 并结束程序
    if (!graph.topologicalSort())
    {
        cout << 0 << endl;
        return 0;
    }

    // 计算最早开始时间和最晚开始时间
    graph.calculateTimes();
    // 找出并输出所有关键活动
    graph.findCriticalActivities();

    return 0;
}
```

4 项目测试

4.1 正常测试

4.1.1 简单情况测试

测试用例： 7 8

1 2 4

1 3 3

2 4 5

3 4 3

4 5 2

4 6 6

5 7 5

6 7 2

预期结果： 17

1->2

2->4

4->6

6->7

测试结果：

```
请输入任务交接点数量和子任务数量：7 8
请依次输入任务的开始和完成的交接点编号以及所需时间：
1 2 4
1 3 3
2 4 5
3 4 3
4 5 2
4 6 6
5 7 5
6 7 2
项目总时间：17
关键活动：
1 -> 2
2 -> 4
4 -> 6
6 -> 7
```

4.1.2 一般情况测试，单个起点和单个终点

测试用例： 9 11

1 2 6

1 3 4

1 4 5

2 5 1

3 5 1

4 6 2

5 7 9

5 8 7

6 8 4

7 9 2

8 9 4

预期结果： 18

1->2

2->5

5->8

5->7

7->9

8->9

测试结果：

```
请输入任务交接点数量和子任务数量： 9 11
请依次输入任务的开始和完成的交接点编号以及所需时间：
1 2 6
1 3 4
1 4 5
2 5 1
3 5 1
4 6 2
5 7 9
5 8 7
6 8 4
7 9 2
8 9 4
项目总时间： 18
关键活动：
1 -> 2
2 -> 5
5 -> 8
5 -> 7
7 -> 9
8 -> 9
```

4.1.3 不可行的方案测试

测试用例： 4 5

1 2 4

2 3 5

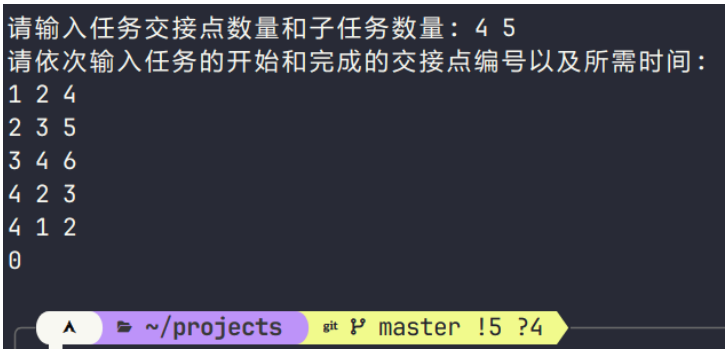
3 4 6

4 2 3

4 1 2

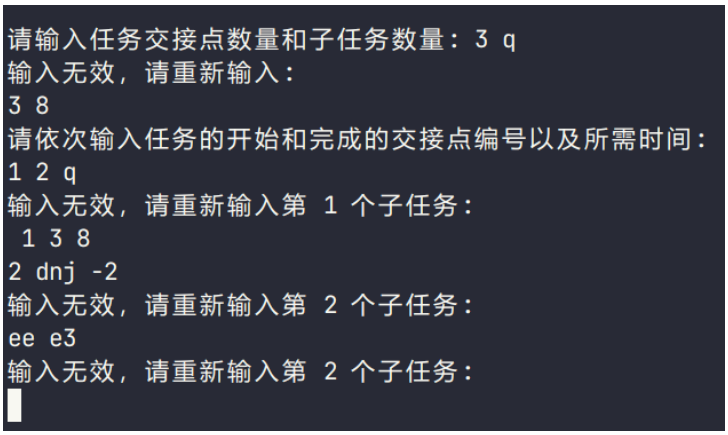
预期结果： 0

测试结果：



4.2 健壮性测试

程序健壮性表现良好，对于不合法的输入会给出提示，并要求重新输入。



5 项目心得与体会

在完成关键任务项目——任务调度与关键路径分析系统的开发过程中，我获得了宝贵的实践经验和深刻的认识，这些经验不仅增强了我的专业知识，也提升了我的实践技能和解决问题的能力。

通过本项目，我深刻体会到了数据结构和算法在解决实际问题中的重要性。在设计 Edge、Queue、Graph 等核心类的过程中，我不仅复习了基本的数据结构知识，还学会了如何将这些数据结构应用于解决实际问题。特别是在实现拓扑排序和关键路径分析算法时，我学会了如何利用队列和图的结构来高效地处理任务依赖关系和计算任务的执行顺序；拓扑排序和关键路径分析是图论中的经典问题，它们的实现让我对图的遍历、排序和搜索有了更加深入的认识；在编写和调试代码的过程中，我的编程能力得到了显著提升。我学会了如何编写清晰、高效且易于维护的代码，并且掌握了一些调试技巧，这些都对我的未来编程工作大有裨益；本项目让我学会了如何将复杂的理论问题转化为具体的编程任务，并找到有效的解决方案。我学会了如何规划项目、分解任务、分配资源，并确保项目按时完成。这些经验和技能将为我未来的学习和工作奠定坚实的基础。

总之，这个项目不仅让我掌握了数据结构和算法的具体应用，还提升了我的编程能力、问题解决能力和系统设计能力。这些经验和技能将为我未来的学习和工作奠定坚实的基础。