项目说明文档

数据结构课程设计

——关键活动

作 者 姓 名：张翔

学　　号：2352985

指 导 教 师：张颖

学院、 专业：计算机科学与技术学院 软件工程

同济大学

Tongji University

二〇二四 年 十二 月 七 日

1 项目分析

1.1 项目背景分析

任务调度问题广泛应用于工程项目管理、生产调度和资源分配等领域，其核心在于通过合理安排任务顺序和分配资源，优化整个工程的工期与资源使用效率。在实际工程中，各任务之间通常存在一定的优先关系，部分任务的延迟可能对全局进度产生重大影响。关键路径法（CPM）是一种经典的时间优化方法，它能够识别出那些直接影响工期的关键任务。通过分析任务的依赖关系和时间需求，识别关键活动不仅有助于项目管理者优先关注这些任务，也为调整资源分配提供了决策依据。

1.2 项目需求分析

该实验需要解决工程项目的关键活动判定问题，以保证任务调度的合理性。具体而言，需要判断给定的任务调度是否可行，即是否符合任务的优先约束条件。如果调度方案可行，则计算完成整个项目的最短时间，并输出所有关键活动，以明确那些需要优先关注的任务。同时，实验还需考虑到输入任务数据的有效性和调度过程的鲁棒性，确保算法在处理较复杂任务关系时的正确性和效率。此外，输出的关键活动应清晰展示其在全局调度中的重要性，以便管理者快速了解项目的风险点。

1.3 项目功能分析

程序的核心功能包括以下几点：第一，解析输入任务的依赖关系和所需时间，构建任务的有向图模型；第二，基于拓扑排序和关键路径算法，验证调度的可行性并计算完成整个工程所需的最短时间；第三，分析每个任务的早最时间、晚最时间以及自由浮动时间，从而识别出所有的关键活动；第四，输出调度的整体结果，包括最短完成时间、关键活动及其路径。同时，程序需要具备良好的异常处理能力，能够应对如输入数据错误或图中存在环等问题，以保证实验的健壮性和科学性。

2 项目设计

2.1 数据结构设计

在实现关键活动项目时，数据结构的设计主要依赖于有向图（Directed Graph），用于表示任务之间的依赖关系。在该图中，每个任务作为一个顶点，任务之间的依赖关系则通过边表示，边的权重代表完成任务所需的时间。关键活动识别依赖于图的拓扑排序过程，通过计算每个任务的最早开始时间（earliest start time）和最晚完成时间（latest finish time），可以判断哪些任务属于关键活动——即那些如果延误，将直接影响整个项目工期的任务。为了实现这一过程，图结构需要支持顶点和边的添加、查找和排序操作。通过拓扑排序获取任务执行的顺序，进而可以进行时间

计算，最终确定关键活动。此外，图的空间复杂度和时间复杂度与任务数量和依赖关系的数量成正比，确保在规模较大的项目中仍能有效处理任务调度和关键活动识别的问题。

2.2 MyDirectedGraph 类的设计

2.2.1 概述

MyDirectedGraph 是一个模板类，用于表示有向图的基本操作和关键活动计算。

它包括最大顶点数、当前顶点数、边数、顶点数组和邻接矩阵等成员变量，用于存储图的结构。

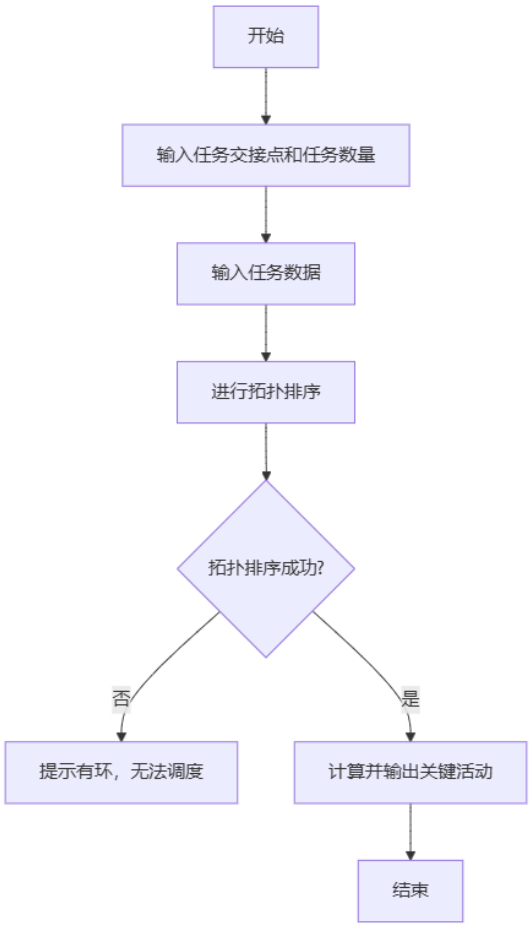
该类提供了基本的图操作功能，如添加顶点、添加边、查找边及获取顶点和边的数量。同时，类还实现了拓扑排序和关键活动的计算功能。topologicalSort 方法用于判断图是否可进行拓扑排序，若排序成功，则通过 printCriticalActivities 方法输出关键活动。该类支持任意类型的顶点，并通过邻接矩阵实现图的存储结构。

2.2.2 类定义

```
template <typename Type>
class MyDirectedGraph
{
private:
    int maxVertices;
    int vertexCount;
    int edgeCount;
    Type* vertices;
    Edge** graph;
    int findVertexIndex(const Type& vertex) const;
public:
    // 基础功能
    MyDirectedGraph(int _maxVertices);
    ~MyDirectedGraph();
    int getVertexCount() { return vertexCount; }
    int getEdgeCount() { return edgeCount; }
    bool addVertex(const Type& vertex);
    bool addEdge(const Type& vertexA, const Type& vertexB, int weight);
    bool findEdge(const Type& vertexA, const Type& vertexB);
```

```
    // 关键活动
    bool topologicalSort();
    void printCriticalActivities();
};
```

2.3 项目主题架构设计



2.3.1 项目主题架构设计流程图

3 项目功能实现

3.1 项目主题架构实现

3.2.1 项目主题架构实现思路

这段代码实现了一个任务调度管理系统的核心功能，旨在根据用户输入的任务交接点和任务信息，判断任务调度的可行性并输出关键活动，其具体实现思路如下：

(1)程序提示用户输入任务交接点的数量和任务数量，并根据交接点数量初始化有向图结构。

(2)用户依次输入每个任务的起点、终点及所需时长，程序对输入数据进行合法性检查，确保交接点编号在指定范围内且任务时长为正整数；若输入无效，则提示错误并重新输入。

(3)所有任务数据录入完成后，程序通过拓扑排序验证调度的可行性。如果存在环路，则说明任务调度无解；若拓扑排序成功，则进一步计算任务的关键活动，并输出其详细信息。

整个实现逻辑清晰，功能模块化，能够有效处理任务调度中的核心问题。

3.2.2 项目主题架构核心代码

```
int main()
{
    /* 系统进入提示语 */
    std::cout << ">>> 欢迎使用任务调度管理系统" << std::endl << std::endl;

    /* 输入任务交接点和任务的数量 */
    int N = inputInteger(1, MAX_TASK_HANDOVER, "请输入任务交接点的数量");
    int M = inputInteger(1, MAX_TASK, "请输入任务的数量");

    /* 初始化有向图 */
    MyDirectedGraph<int> graph(N);
    for (int i = 1; i <= N; i++)
        graph.addVertex(i);

    std::cout << ">>> 请输入每个任务的交接点及任务所需时长(请按照顺序输入): " <<
std::endl;
    std::cout << ">>> 对于每个任务，请输入两个交接点的编号（从 1 开始）以及该任务的
时长" << std::endl;
    std::cout << ">>> 例如，输入“1 2 5”表示任务从交接点 1 到交接点 2，任务所需时长
为 5 单位" << std::endl << std::endl;

    for (int i = 0; i < M; i++) {
        int from, to, weight;
        std::cout << "任务" << std::setw(3) << i + 1 << ": ";
        std::cin >> from >> to >> weight;
```

```
            if (from <= 0 || from > N || to <= 0 || to > N || weight <= 0) {
                std::cout << std::endl << ">>> 输入无效，请确保交接点编号在范围内，并且任务时长为正整数" << std::endl;
                i--;
                continue;
            }
            graph.addEdge(from, to, weight);
        }

        std::cout << std::endl;
        std::cout << ">>> 正在进行拓扑排序..." << std::endl;

        if (!graph.topologicalSort()) {
            std::cout << ">>> 无法完成拓扑排序: 存在环路，无法完成任务调度 " << std::endl;
        }
        else {
            std::cout << ">>> 拓扑排序成功，正在计算关键活动..." << std::endl << std::endl;
            graph.printCriticalActivities();
        }

        // 退出程序
        return 0;
    }
```

3.2 拓扑排序功能的实现

3.2.1 拓扑排序功能实现思路

这段拓扑排序的实现通过 Kahn 算法（基于入度法）来对有向图进行排序。首先，计算每个节点的入度，即统计每个节点指向的其他节点的数量，存储在 inDegree 数组中。然后，找到所有入度为 0 的节点，这些节点没有前驱，可以作为拓扑排序的起点。将这些节点加入栈中，开始处理。在处理每个节点时，遍历与之相连的节点，并将它们的入度减 1。如果某个节点的入度变为 0，则将其加

入栈中，继续处理。整个过程中，维护一个计数器 processedCount，记录已经处理的节点数。最后，如果处理的节点数等于图中的总节点数，说明图中没有环，拓扑排序成功；否则，图中存在环，排序无法完成。该算法使用了栈来存储入度为 0 的节点，确保能够按拓扑顺序输出节点。

3.2.2 拓扑排序功能核心代码

```
template <typename Type>
bool MyDirectedGraph<Type>::topologicalSort()
{
    // 存储每个节点的入度
    int* inDegree = new(std::nothrow) int[maxVertices];
    if (inDegree == nullptr) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(-1);
    }
    for (int i = 0; i < maxVertices; i++) {
        inDegree[i] = 0;
    }

    for (int i = 0; i < maxVertices; i++) {
        for (int j = 0; j < maxVertices; j++) {
            if (graph[i][j].exist) { // 如果存在边，从 i 到 j
                inDegree[j]++;  // 增加 j 的入度
            }
        }
    }

    // 使用栈保存入度为 0 的节点
    int* stack = new int[maxVertices];
    int top = -1;
    for (int i = 0; i < maxVertices; i++) {
        if (inDegree[i] == 0) {
            stack[++top] = i;
```

```
        }
    }

    int processedCount = 0; // 记录已处理的节点数

    while (top != -1) {
        int node = stack[top--];
        processedCount++;
        for (int i = 0; i < maxVertices; i++) {
            if (graph[node][i].exist) {  // 如果有边 node -> i
                if (--inDegree[i] == 0) {  // 入度减 1，如果变为 0，则可以加入栈中
                    stack[++top] = i;
                }
            }
        }
    }

    // 如果处理的节点数少于图中的总节点数，则说明存在环，不能完成拓扑排序
    if (processedCount != vertexCount) {
        std::cout << "Error: The graph contains a cycle, topological sort is not
possible." << std::endl;
        return false;
    }

    delete[] inDegree;
    delete[] stack;
    return true;
}
```

3.3 关键活动寻找功能的实现

3.3.1 关键活动寻找功能实现思路

该函数的实现思路通过计算最早开始时间和最晚开始时间来确定关键活动。首先，初始化两个数组 earliestStart 和 latestStart，分别用于存储每个节点的最早开始时间和最晚开始时间。最早开始时间从源点开始计算，遍历图中的每一条边，更新目标节点的最早开始时间。接下来，最晚开始时间从汇点开始计算，逆序遍历图中的边，更新源节点的最晚开始时间。关键活动的判定基于这样的条件：如果某个任务的最早开始时间等于其后续任务的最晚开始时间减去任务时长，则该任务为关键活动。最后，遍历所有任务，输出满足条件的关键活动及其最早开始时间。

3.3.2 关键活动寻找功能核心代码

```
template <typename Type>
void MyDirectedGraph<Type>::printCriticalActivities()
{
    int* earliestStart = new int[maxVertices];
    int* latestStart = new int[maxVertices];
    if (earliestStart == nullptr || latestStart == nullptr) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(-1);
    }

    // 初始化最早开始时间和最晚开始时间
    for (int i = 0; i < maxVertices; i++) {
        earliestStart[i] = 0;
        latestStart[i] = INT_MAX;
    }

    // 计算最早开始时间（从源点开始）
    for (int i = 0; i < maxVertices; i++) {
        for (int j = 0; j < maxVertices; j++) {
            if (graph[i][j].exist) {
                earliestStart[j] = std::max(earliestStart[j], earliestStart[i] +
graph[i][j].weight);
            }
        }
```

```
    }

    // 计算最晚开始时间（从汇点开始）
    latestStart[maxVertices - 1] = earliestStart[maxVertices - 1];
    for (int i = maxVertices - 2; i >= 0; i--) {
        for (int j = 0; j < maxVertices; j++) {
            if (graph[i][j].exist) {
                latestStart[i] = std::min(latestStart[i], latestStart[j] -
graph[i][j].weight);
            }
        }
    }

    // 输出关键活动及最小时间
    std::cout << ">>> 关键活动及最小时间: " << std::endl;
    for (int i = 0; i < maxVertices; i++) {
        for (int j = 0; j < maxVertices; j++) {
            if (graph[i][j].exist && earliestStart[i] == latestStart[j] -
graph[i][j].weight) {
                std::cout << "关键活动: " << vertices[i] << " -> " << vertices[j]
                    << " | 最早开始时间: " << earliestStart[i] << std::endl;
            }
        }
    }

    delete[] earliestStart;
    delete[] latestStart;
}
```

3.3 异常处理功能的实现

3.3.1 动态内存申请失败的异常处理

在进行动态内存申请时，程序使用 new(std::nothrow)来尝试分配内存。new(std::nothrow)在分配内存失败时不会引发异常，而是返回一个空指针（nullptr），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，这时程序将执行以下操作：

(1)向标准错误流 std::cerr 输出一条错误消息"Error: Memory allocation failed."；

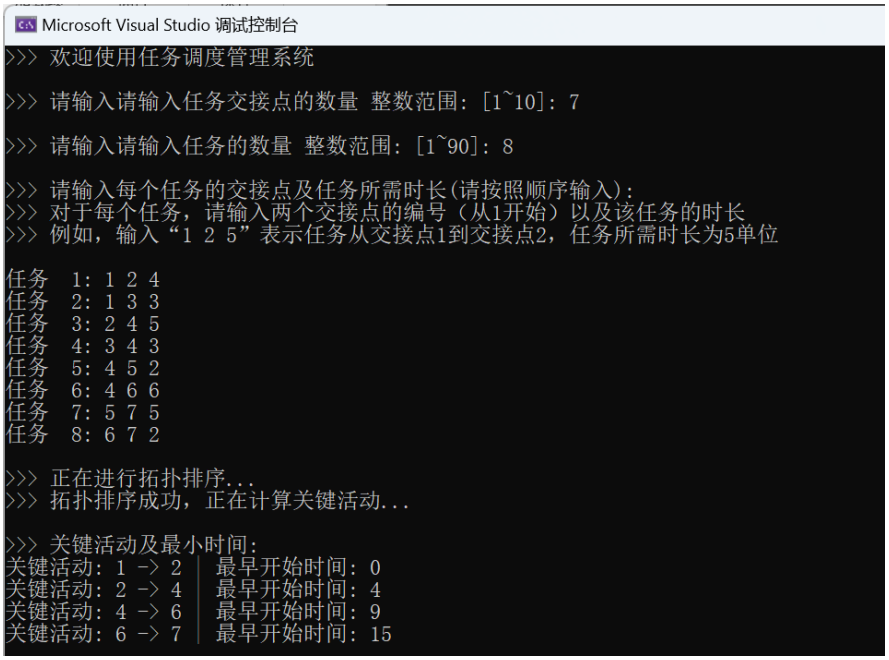
(2)调用 exit 函数，返回错误码-1，用于指示内存分配错误，并导致程序退出。

3.3.2 输入非法的异常处理

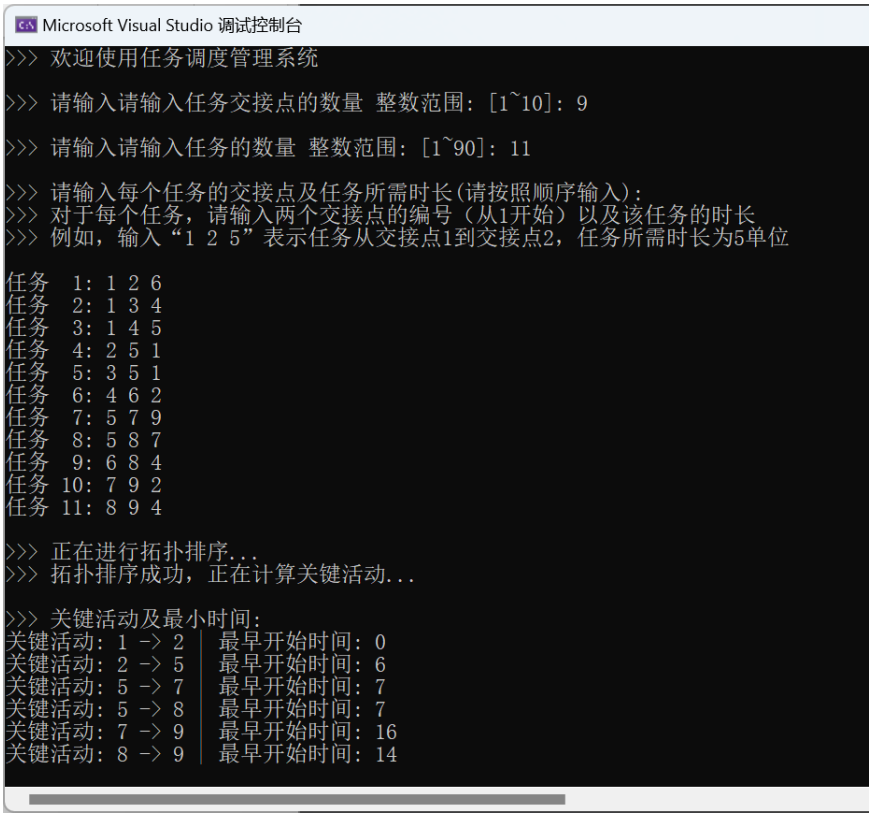
程序通过调用 inputInteger 函数输入任务交接点的数量和任务的数量。inputInteger 函数用于获取用户输入的整数，同时限制输入必须在指定的范围内，函数的代码如下：

```
int inputInteger(int lowerLimit, int upperLimit, const char* prompt)
{
    std::cout << ">>> " << "请输入" << prompt << " 整数范围: [" << lowerLimit << "~" << upperLimit << "]: ";
    int input;
    while (true) {
        std::cin >> input;
        if (std::cin.good() && input >= lowerLimit && input <= upperLimit) {
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
            std::cout << std::endl;
            return input;
        }
        else {
            std::cerr << ">>> " << prompt << "输入不合法，请重新输入！" << std::endl;
            std::cin.clear();
            std::cin.ignore(INT_MAX, '\n');
        }
    }
}
```

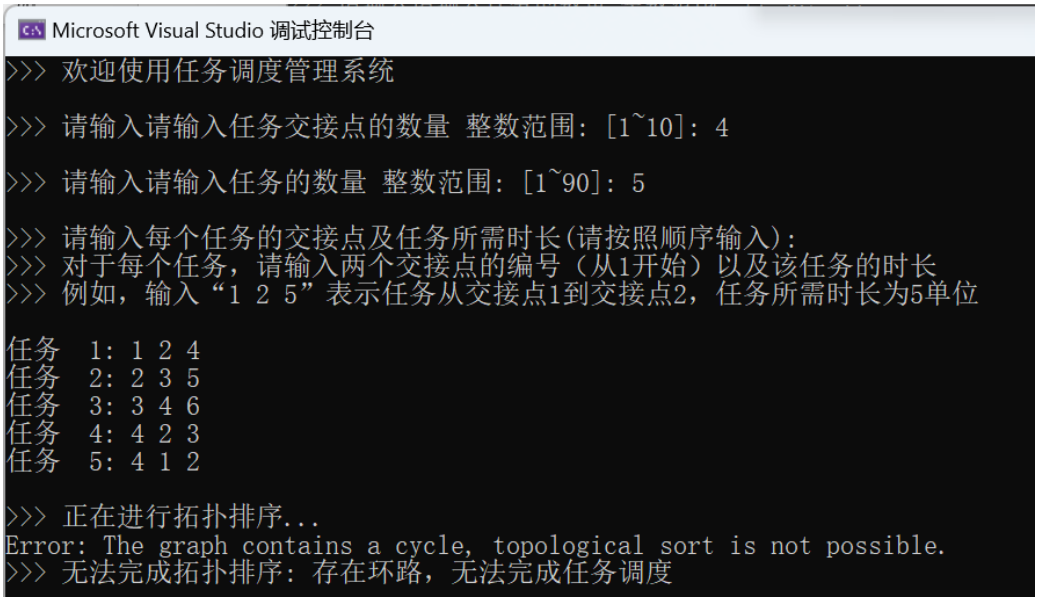
4 项目测试



4.1 简单情况测试示例



4.2 一般情况测试示例



4.3 不可行的方案测试示例

5 集成开发环境与编译运行环境

Windows 系统：Windows 11 x64

Windows 集成开发环境：Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境：本项目适用于 x86 架构和 x64 架构