数据结构课程设计文档

题目：关键活动

姓名：赵卓冰

学号：2252750

专业：软件工程

年级：2023 级

指导教师：张颖

2024 年 12 月 5 日

运行环境与开发工具
问题描述
功能需求
1 输入说明
2 输出说明
项目设计
1 数据结构设计
1. 1 **任务表示**
1. 2 **图的表示**
1. 2. 1 邻接表表示
1. 3 **拓扑排序辅助结构**
1. 3. 1 **入度数组**
1. 3. 2 **队列**
1. 4 **时间信息记录**
1. 4. 1 **最早完成时间数组**
1. 4. 2 **最晚完成时间数组**
1. 5 **关键活动记录**
1. 5. 1 **关键活动集合**
1. 5. 2 **排序关键活动**
1. 6 数据结构关系图
2 算法设计
2. 1 拓扑排序
2. 2 关键路径计算
2. 3 关键活动排序
功能实现
1 **核心函数实现**
1. 1 拓扑排序函数
1. 2 关键路径计算函数
2 流程图
2. 1 流程图各步骤详细说明
测试结果
1 Windows平台
1. 1 测试用例 1
1. 2 测试用例 2
1. 3 测试用例 3
2 Linux平台
2. 1 测试用例 1
2. 2 测试用例 2
2. 3 测试用例 3
心得体会

1. 运行环境与开发工具

本项目支持在以下开发环境和编译运行环境中运行：

- **Windows 操作系统**：
  - 版本：Windows 10 x64

    - IDE：Visual Studio 2022 (Debug 模式)
    - 编译器：MSVC 14.39.33519
- **Linux 操作系统**：
    - 版本：Ubuntu 20.04.6 LTS
    - IDE：VS Code
    - 编译器：gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)

2. 问题描述

在工程项目管理中，任务调度问题是一个重要且复杂的问题。不同子任务之间往往存在依赖关系，任务的完成时间和依赖关系会直接影响整个项目的工期。有效地分析和优化任务调度，不仅可以减少不必要的时间浪费，还能保障项目按期甚至提前完成。

1. **任务依赖关系：**
    - 在工程项目中，不同的子任务可能有前后依赖关系。例如，某个任务必须在其前置任务完成之后才能开始。
    - 任务之间的依赖关系可以用节点和边来表示，其中节点代表任务的交接点，边代表任务，边权是完成该任务需要的时间。
2. **输入任务数据：**
    - 输入包括交接点的数量和任务的数量。
    - 每个任务通过起点交接点和终点交接点定义，并附带完成该任务所需的时间。
3. **输出目标：**
    - 如果输入的任务依赖关系中存在环，则任务调度方案不可行，因为环代表循环依赖，项目无法推进。
    - 如果调度方案可行，程序需要计算完成整个工程项目的最短时间（即项目的**关键路径长度**）。
    - 程序还需要找出所有的**关键活动**，即那些对项目整体工期至关重要的任务。一旦这些任务的完成时间延误，整个项目的工期将随之延误。
4. **关键活动：**
    - 一个任务是否为关键活动，取决于其起点的最早完成时间、终点的最晚开始时间以及任务持续时间。如果满足：

      $$\text{earliest}[start] + \text{duration} = \text{latest}[end]$$

      则该任务为关键活动。
5. **输出要求：**
    - 如果任务调度不可行，直接输出 `0`，表示该调度方案无法执行。
    - 如果调度可行：
        - 输出完成整个项目的最短时间。
        - 按照规则输出所有的关键活动，顺序为：
            1. 起点编号较小的任务优先；

2. 若起点编号相同，则任务按输入顺序的逆序排列。

3. 功能需求

3.1. 输入说明

1. 第1行输入两个正整数 N 和 M：
   - N 是任务交接点（项目依赖的节点）数量，编号为 1 到 N；
   - M 是子任务数量，编号为 1 到 M。
2. 接下来 M 行，每行输入 3 个正整数：
   - **任务的起始交接点编号**；
   - **任务的结束交接点编号**；
   - **完成该任务所需时间。**

3.2. 输出说明

1. 如果任务调度不可行（存在循环依赖），输出 `0`；
2. 否则，第一行输出完成整个项目的最短时间；
3. 第二行开始，输出所有关键活动，格式为 `V->W`，其中 V 和 W 是任务的起始和结束交接点编号。关键活动按以下顺序输出：
   - 起始交接点编号小者优先；
   - 若起点编号相同时，与输入顺序相反。

4. 项目设计

4.1. 数据结构设计

为了高效地表示任务调度问题中的依赖关系及计算结果，需要设计以下数据结构：

4.1.1. 任务表示

任务是调度问题的基本单元，包含以下信息：

- **起始交接点（start）**：任务的起点编号。
- **结束交接点（end）**：任务的终点编号。
- **任务持续时间（duration）**：完成该任务需要的时间。
- **输入顺序索引（index）**：记录任务在输入时的顺序，方便按照需求对关键活动排序。

任务的结构体设计如下：

```cpp
struct Task {
    int start;    // 起点编号
    int end;      // 终点编号
    int duration; // 持续时间
    int index;    // 输入时的索引
};
```

4.1.2. 图的表示

任务之间的依赖关系可以用有向图表示：

- **节点（交接点）**：每个任务的起点或终点。
- **边（任务）**：从任务的起点到终点的有向边，边的权重是任务的持续时间。

4.1.2.1. 邻接表表示

为了节约空间，采用邻接表表示有向图：

- 用 `Vector<Vector<pair<int, int>>>` 表示邻接表。
- 外层 `Vector` 的每个元素表示一个节点的出边集合。
- 内层 `pair<int, int>` 表示一条边，包含两个信息：
  - **终点编号**：该任务的结束交接点编号。
  - **任务权重**：任务的持续时间。

邻接表初始化与存储示例：

```cpp
Vector<Vector<pair<int, int>>> graph;
graph = Vector<Vector<pair<int, int>>>(num_nodes + 1); // 初始化，节点编号从1到N
graph[start].PushBack({end, duration}); // 添加从start到end的任务，权重为duration
```

4.1.3. 拓扑排序辅助结构

4.1.3.1. 入度数组

记录每个节点的入度，用于拓扑排序：

- 节点入度为 0 时，说明该节点没有前置依赖任务，可以作为拓扑排序的起始节点。

入度数组的定义与初始化：

```cpp
Vector<int> in_degree(num_nodes + 1, 0); // 节点编号从1到N，初始入度为0
in_degree[end]++; // 每次添加一条边时，增加终点节点的入度
```

4.1.3.2. 队列

在拓扑排序过程中，用队列记录所有入度为 0 的节点，逐一处理它们的出边，动态更新依赖关系。

队列初始化与操作：

```
Queue<int> q;
q.Push(node); // 入队
q.Pop(); // 出队
int front_node = q.Front(); // 获取队首节点
```

4.1.4. 时间信息记录

4.1.4.1. 最早完成时间数组

记录每个节点的最早完成时间，表示在拓扑排序中计算出的从起点到该节点的最长路径长度：

- 对于起点，最早完成时间为 0；
- 对于每个任务的终点：

$$\text{earliest}[end] = \max(\text{earliest}[end], \text{earliest}[start] + \text{duration})$$

最早完成时间数组定义与初始化：

```
Vector<int> earliest(num_nodes + 1, 0); // 初始最早完成时间为0
```

4.1.4.2. 最晚完成时间数组

记录每个节点的最晚完成时间，表示项目总时间确定后，倒序计算出的从终点到该节点的最长路径长度：

- 对于终点，最晚完成时间为项目总时间；
- 对于每个任务的起点：

$$\text{latest}[start] = \min(\text{latest}[start], \text{latest}[end] - \text{duration})$$

最晚完成时间数组定义与初始化：

```
Vector<int> latest(num_nodes + 1, INT_MAX); // 初始最晚完成时间为无穷大
latest[end] = project_time; // 项目总时间设置为终点的最晚完成时间
```

4.1.5. 关键活动记录

4.1.5.1. 关键活动集合

用一个 `Vector` 记录所有关键活动的索引：

- 遍历所有任务，检查以下条件：

$$\text{earliest}[start] + \text{duration} = \text{latest}[end]$$

- 如果条件满足，说明该任务是关键活动。

关键活动集合定义与添加：

```
Vector<int> critical_activities;
if (earliest[start] + duration == latest[end]) {
    critical_activities.PushBack(task_index);
}
```

4.1.5.2. 排序关键活动

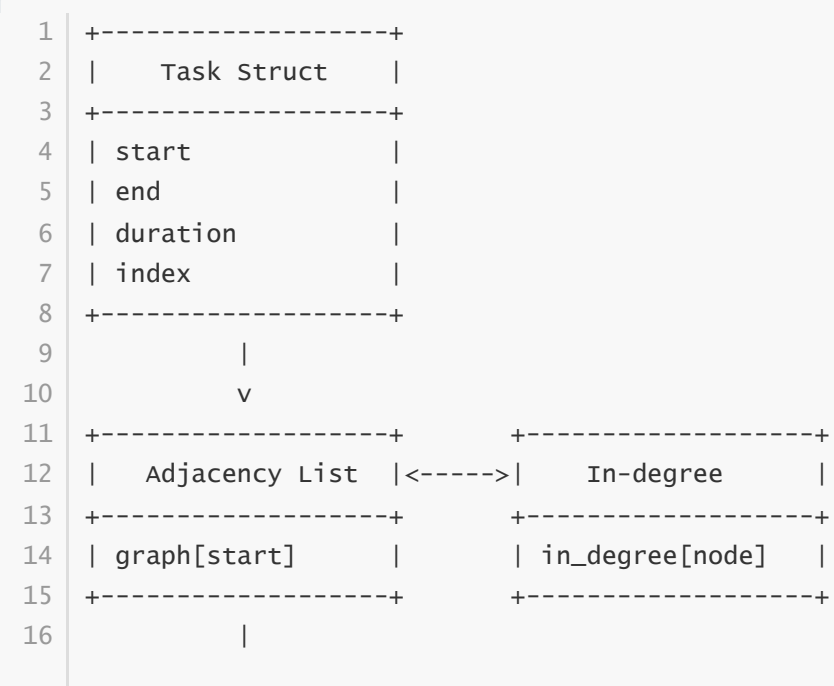
排序规则：

1. 起点编号小者优先；
2. 起点编号相同时，按输入顺序的逆序排列。

排序实现：

```
// 冒泡排序
size_t n = critical_activities.Size();
for (size_t i = 0; i < n; ++i) {
    for (size_t j = 0; j < n - i - 1; ++j) {
        const auto& task_a = tasks[critical_activities[j]];
        const auto& task_b = tasks[critical_activities[j + 1]];
        if (task_a.start > task_b.start ||
            (task_a.start == task_b.start && task_a.index < task_b.index)) {
            swap(critical_activities[j], critical_activities[j + 1]);
        }
    }
}
```

4.1.6. 数据结构关系图



```
          v
+-------------------+       +-------------------+
| Earliest Times    |<----->| Latest Times      |
+-------------------+       +-------------------+
| earliest[node]    |       | latest[node]      |
+-------------------+       +-------------------+
          |
          v
+-------------------+
| Critical Activities|
+-------------------+
| critical_activities|
+-------------------+
```

这些数据结构相互协作，共同完成任务调度的依赖管理和关键路径的计算。

4.2. 算法设计

4.2.1. 拓扑排序

通过拓扑排序判断任务调度是否可行，并计算各节点的最早完成时间。

步骤：

1. 初始化入度数组；
2. 将所有入度为 `0` 的节点加入队列；
3. 依次处理队列中的节点：
    - 更新相邻节点的入度；
    - 更新相邻节点的最早完成时间；
    - 若相邻节点入度变为 `0`，将其加入队列。
4. 如果处理节点数等于总节点数，说明任务调度可行；否则，调度不可行（存在环）。

4.2.2. 关键路径计算

基于拓扑排序结果，计算各节点的最晚完成时间，并判定关键活动。

步骤：

1. 初始化所有节点的最晚完成时间为项目总时长；
2. 按拓扑排序的逆序处理每个节点，更新其前驱节点的最晚完成时间；
3. 遍历所有任务，判定是否满足关键活动条件：
   $\text{earliest}[start] + \text{duration} = \text{latest}[end]$

4.2.3. 关键活动排序

使用冒泡排序对关键活动按要求排序：

1. 起点编号小者优先；
2. 起点编号相同时，按输入顺序的逆序排列。

5. 功能实现

5.1. 核心函数实现

5.1.1. 拓扑排序函数

```
bool TopologicalSort(Vector<int>& topo_order) {
    Queue<int> q;

    // 入度为 0 的节点入队
    for (int i = 1; i <= num_nodes; ++i) {
        if (in_degree[i] == 0) {
            q.Push(i);
        }
    }

    while (!q.Empty()) {
        int node = q.Front();
        q.Pop();
        topo_order.PushBack(node);

        for (auto& edge : graph[node]) {
            int next = edge.first;
            int duration = edge.second;

            // 更新最早完成时间
            earliest[next] = max(earliest[next], earliest[node] +
duration);

            // 更新入度
            in_degree[next]--;
            if (in_degree[next] == 0) {
                q.Push(next);
            }
        }
    }

    return topo_order.Size() == num_nodes; // 是否所有节点被处理
}
```

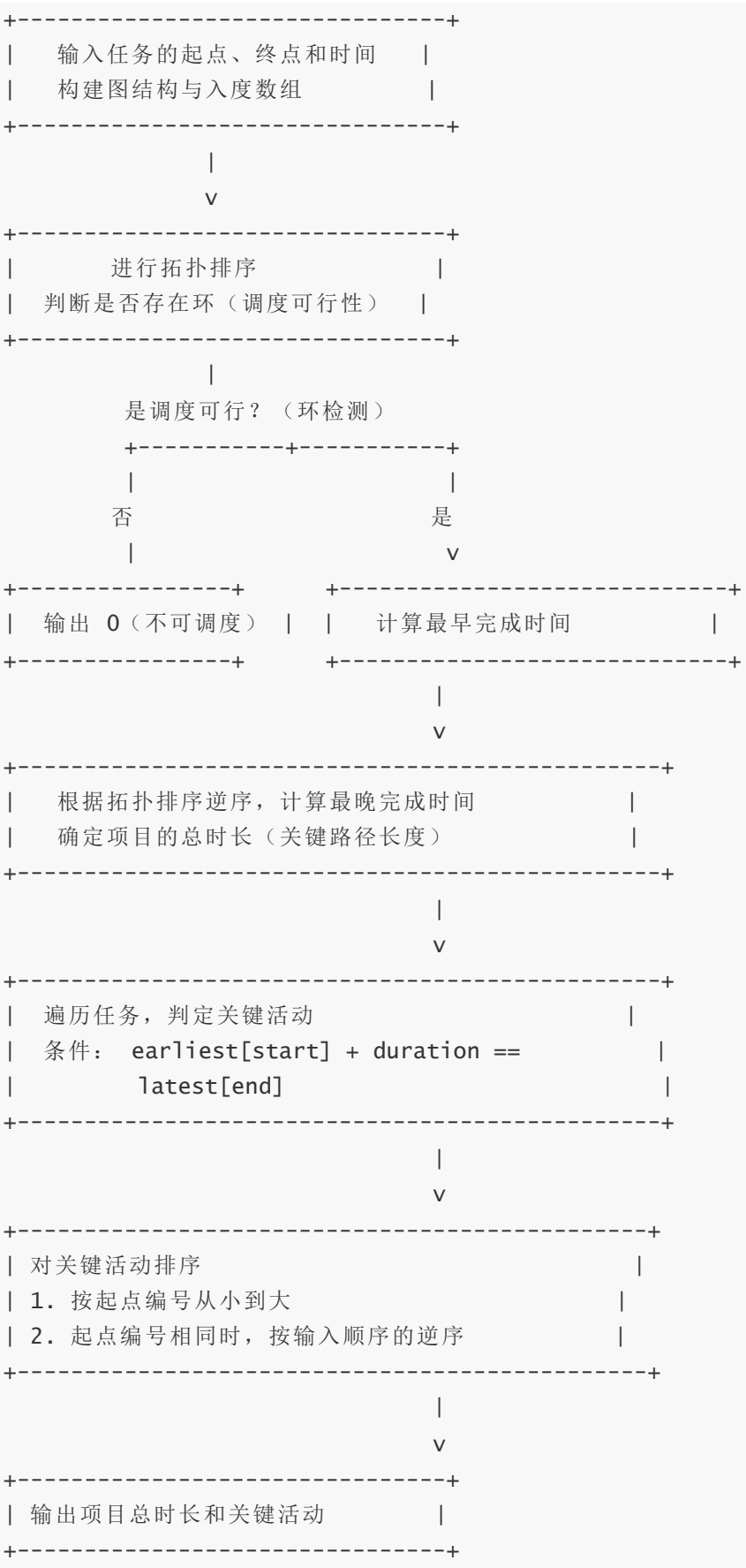
5.1.2. 关键路径计算函数

```
void CalculateCriticalPath() {
    Vector<int> topo_order;

    // 拓扑排序判断任务调度是否可行
    if (!TopologicalSort(topo_order)) {
        cout << 0 << endl; // 不可调度
        return;
```

```
    }

    // 项目总时长
    int project_time = 0;
    for (auto time : earliest) {
        project_time = max(project_time, time);
    }
    latest[topo_order[topo_order.Size() - 1]] = project_time;

    // 倒序计算最晚完成时间
    for (int i = topo_order.Size() - 1; i >= 0; --i) {
        int node = topo_order[i];
        for (auto& edge : graph[node]) {
            int next = edge.first;
            int duration = edge.second;
            latest[node] = min(latest[node], latest[next] - duration);
        }
    }

    // 判定关键活动
    Vector<int> critical_activities;
    for (size_t i = 0; i < tasks.Size(); ++i) {
        const auto& task = tasks[i];
        if (earliest[task.start] + task.duration == latest[task.end]) {
            critical_activities.PushBack(i);
        }
    }

    // 排序关键活动
    SortCriticalActivities(critical_activities);

    // 输出结果
    cout << project_time << endl;
    for (int index : critical_activities) {
        const auto& task = tasks[index];
        cout << task.start << "->" << task.end << endl;
    }
}
```

5.2. 流程图

```
+---------------------------------+
|          程序启动               |
+---------------------------------+
                |
                v
+---------------------------------+
| 输入交接点数量 N 和任务数量 M   |
+---------------------------------+
                |
                v
```


输入任务的起点、终点和时间
构建图结构与入度数组
进行拓扑排序
判断是否存在环（调度可行性）
是调度可行？（环检测）
否
是
输出 0（不可调度）
计算最早完成时间
根据拓扑排序逆序，计算最晚完成时间
确定项目的总时长（关键路径长度）
遍历任务，判定关键活动
条件： earliest[start] + duration == latest[end]
对关键活动排序
1. 按起点编号从小到大
2. 起点编号相同时，按输入顺序的逆序
输出项目总时长和关键活动

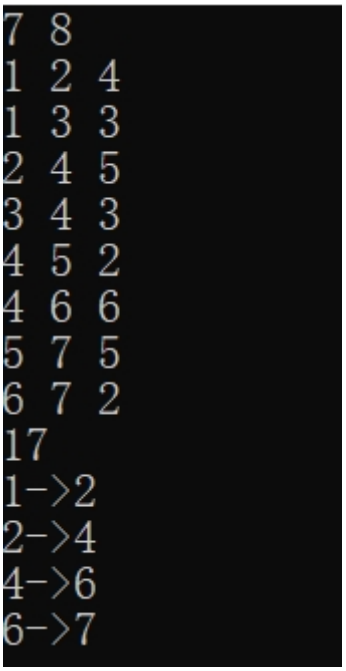
5.2.1. 流程图各步骤详细说明

1. **输入交接点数量和任务数量：**
   - 用户输入节点数量 N 和任务数量 M。
   - 程序根据 N 初始化图、入度数组等数据结构。
2. **输入任务信息：**
   - 用户输入 M 行数据，每行包含任务起点、终点、持续时间。
   - 构建图的邻接表表示，并更新入度数组。
3. **拓扑排序与环检测：**
   - 使用队列进行拓扑排序，按照依赖顺序整理任务。
   - 如果排序完成的节点数小于总节点数 N，说明存在环，输出 `0` 表示调度不可行。
4. **计算最早完成时间：**
   - 根据拓扑排序，逐步更新每个节点的最早完成时间。
5. **计算最晚完成时间：**
   - 根据拓扑排序的逆序，从项目终点向前更新最晚完成时间。
6. **判定关键活动：**
   - 遍历每个任务，根据最早和最晚完成时间的差异，判断任务是否为关键活动。
7. **排序关键活动：**
   - 对所有关键活动按照规则排序：
     - 起点编号小者优先；
     - 起点编号相同时，按输入顺序的逆序排列。
8. **输出结果：**
   - 输出项目总时长。
   - 按顺序输出所有关键活动。

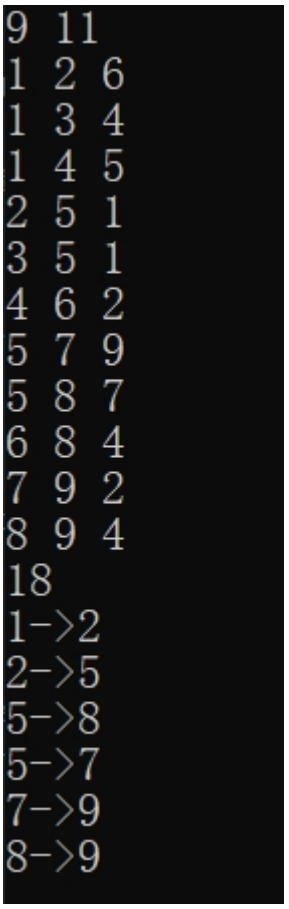
6. 测试结果

6.1. Windows平台

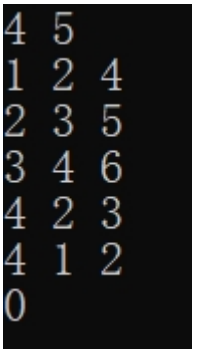
6.1.1. 测试用例 1


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

6.1.2. 测试用例 2

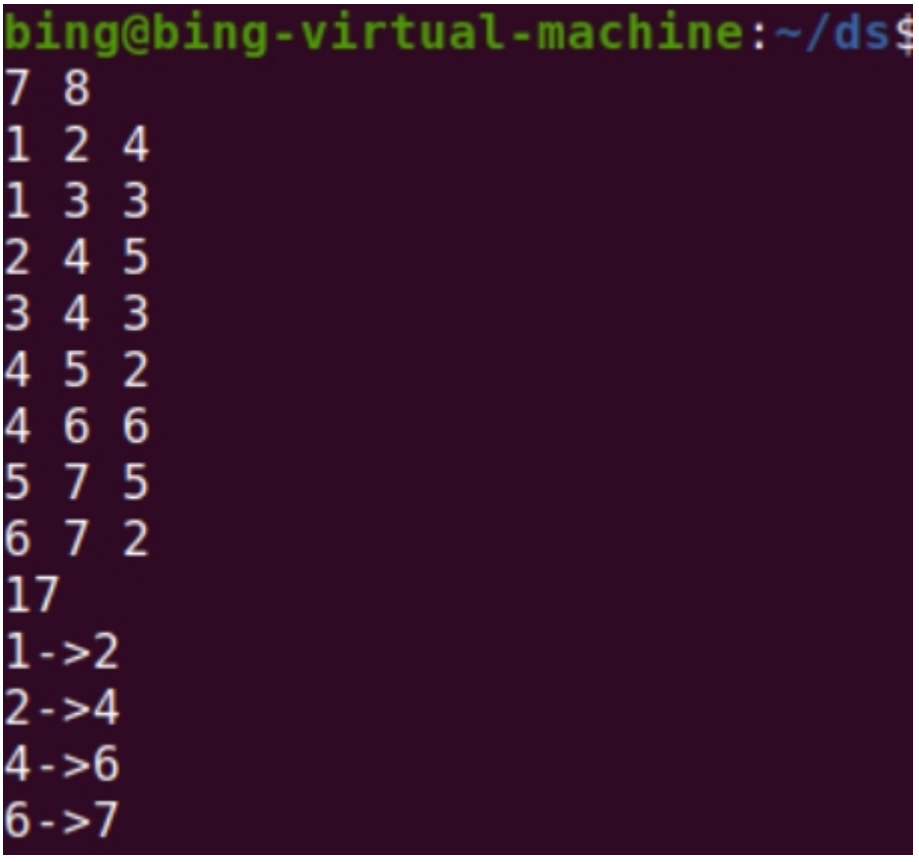

9 11
1 2 6
1 3 4
1 4 5
2 5 1
3 5 1
4 6 2
5 7 9
5 8 7
6 8 4
7 9 2
8 9 4
18
1->2
2->5
5->8
5->7
7->9
8->9

6.1.3. 测试用例 3

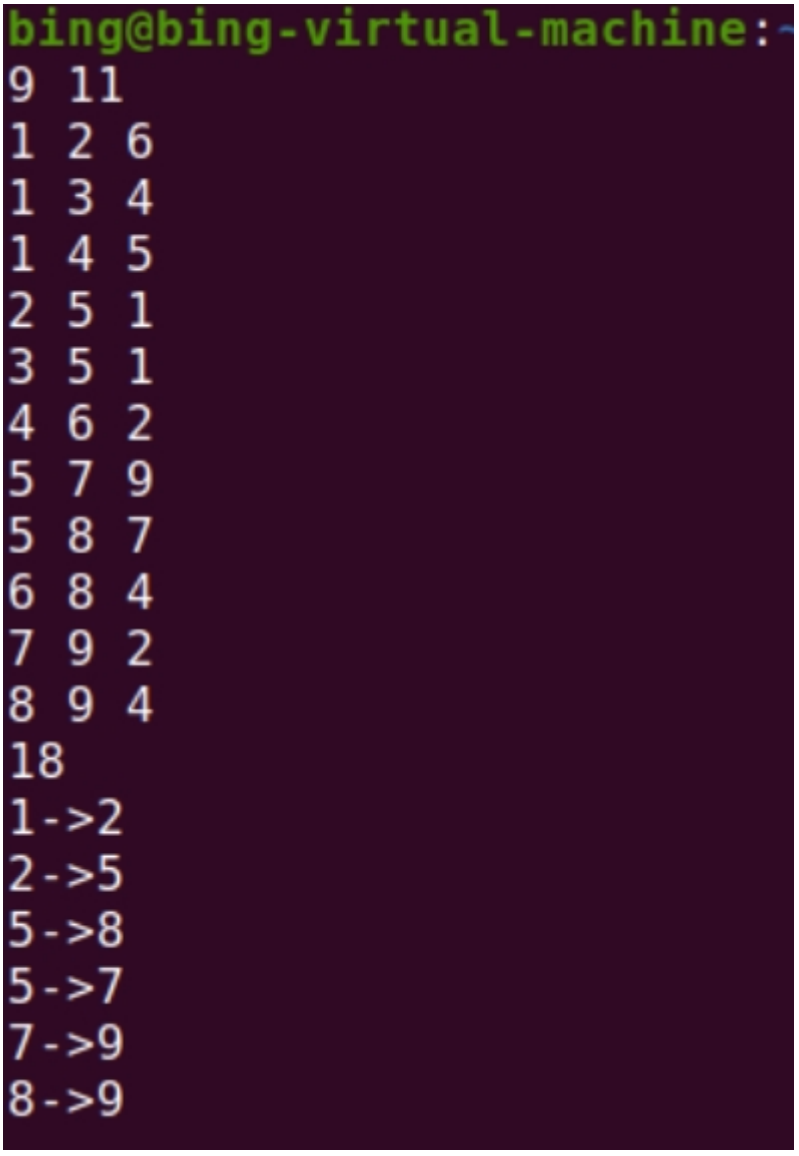


6.2. Linux平台

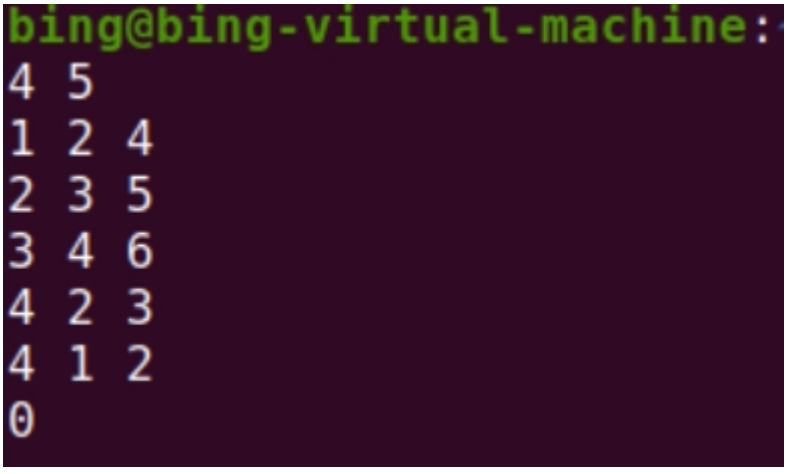
6.2.1. 测试用例 1



6.2.2. 测试用例 2



6.2.3. 测试用例 3



7. 心得体会

通过本实验项目，我加深了对任务调度问题及关键路径算法的理解。在实现过程中：

1. 深刻认识到拓扑排序在检测图中环的有效性；

2. 学会了如何使用最早和最晚完成时间计算关键路径；

3. 理解了如何通过任务的依赖关系构造图并有效存储边权信息。

本项目对算法设计能力和代码实现能力都有较大提升，同时进一步体会了任务调度优化在工程项目管理中的重要性。