实验四 图算法

PB20000137 李远航

一、实验内容及要求

- `Johnson` 算法
  - 实现求所有点对最短路径的 `Johnson` 算法。有向图的顶点数 `N` 的取值分 别为: 27、81、243、729，每个顶点作为起点引出的边的条数取值分别 为：$\log_5 N$、$\log_7 N$（取下整）。图的输入规模总共有4*2=8个，若同一个 `N`，边的两种规模取值相等，则按后面输出要求输出两次，并在报告里说明。(不允许多重边，可以有环。)

二、实验设备及环境

```
OS: Ubuntu 20.04 focal(on the Windows Subsystem for Linux)
Kernel: x86_64 Linux 5.10.102.1-microsoft-standard-WSL2
CPU: Intel Core i5-10200H @ 8x 2.4GHz
GPU: NVIDIA GeForce GTX 1650 Ti
g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
```

三、实验方法和步骤

1. 数据结构设计
   - 使用邻接表存储图的相关数据
2. 判断负环
   - 使用 `spfa` 算法，通过 `dfs`，如果搜索到了已经访问过的节点，说明成负环，则删去路径上的一条边，重新开始搜索

   ```cpp
   void spfa(int u)
   {
       vis[u] = 1;
       for (int i = head[u], v, last = 0; v = edge[i].to, i; last = i, i = edge[i].next)
       {
           if (dis[v] > dis[u] + edge[i].weight)
           {
               dis[v] = dis[u] + edge[i].weight;
               if (vis[v])
               {
                   flag = 1;
                   edge[last].next = edge[i].next;
                   return;
               }
               else
                   spfa(v);
           }
       }
       vis[u] = 0;
   }
   ```
3. `Johnson` 算法

- 构建新图G'

```
for (int i = 1; i <= num; i++)
{
    edge[++cnt].to = i;
    edge[cnt].next = head[num + 1];
    edge[cnt].weight = 0;
    head[num + 1] = cnt;
}
```

- 通过 `Bellman-Ford` 算法计算 `h` 和更新权重

```
for (int i = 1; i <= num; i++)
{
    for (int j = 1; j <= num + 1; j++)
    {
        for (int p = head[j]; p; p = edge[p].next)
        {
            if (h[edge[p].to] > h[j] + edge[p].weight)
                h[edge[p].to] = h[j] + edge[p].weight;
        }
    }
}
```

- 通过优先队列实现 `dijkstra` 算法

```
struct Node
{
    int val;
    int dist;
    Node(int a, int b)
    {
        val = a;
        dist = b;
    }
    friend bool operator<(const struct Node &a, const struct Node &b)
    {
        return a.dist > b.dist;
    }
};
void dijkstra(int u)
{
    std::priority_queue<struct Node> q;
    q.push(Node(u, 0));
    while (!q.empty())
    {
        auto tmp = q.top();
        q.pop();
        int now = tmp.val;
        if (vis[now] == 1)
            continue;
        vis[now] = 1;
        for (int p = head[now]; p; p = edge[p].next)
        {
            if (!vis[edge[p].to] && dis[edge[p].to] > dis[now] +
edge[p].weight)
```

```
                {
                    dis[edge[p].to] = dis[now] + edge[p].weight;
                    Pi[edge[p].to] = now;
                    q.push(Node(edge[p].to, dis[edge[p].to]));
                }
            }
        }
    }
```

- 回溯输出最短路径

```
std::string find_road(int u, int v, int val)
{
    std::string ans = std::to_string(v);
    v = Pi[v];
    if (v == 0)
        return "(" + std::to_string(u) + "," + ans + " NULL)";
    while (u != v)
    {
        ans = std::to_string(v) + "," + ans;
        v = Pi[v];
    }
    return "(" + std::to_string(u) + "," + ans + " " + std::to_string(val) + ")";
}
```

四、实验结果和分析

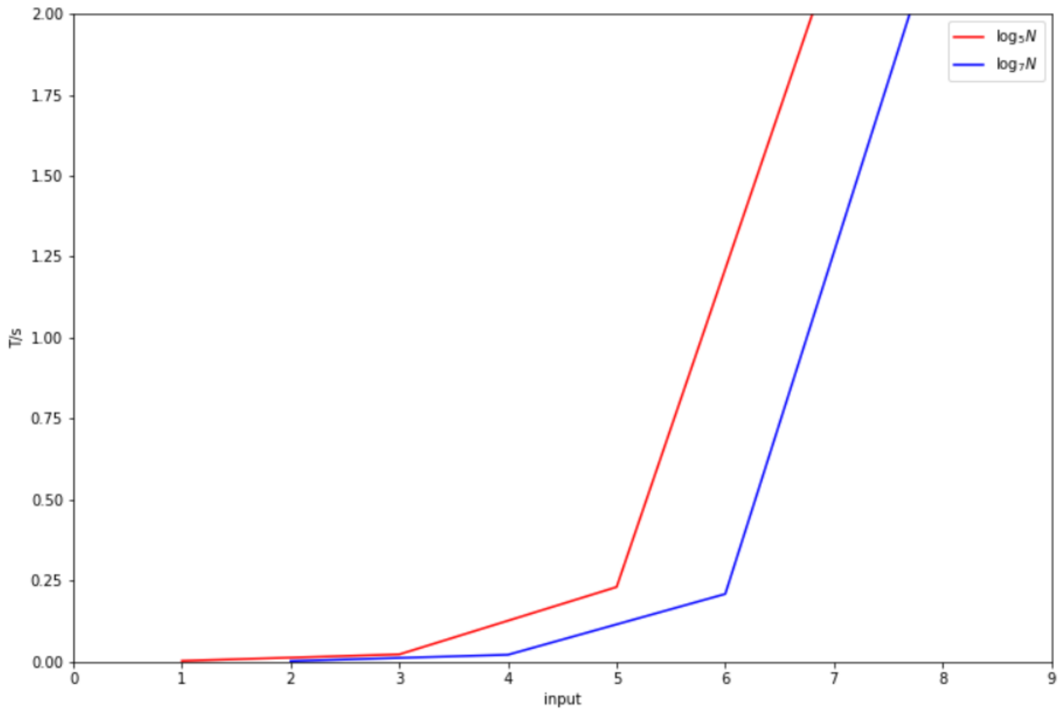
1. 理论复杂度分析

   `c++` 中 `stl` 库的优先队列是通过二叉堆实现的，此时 `dijkstra` 算法的复杂度为$O(E\lg V)$，而 `Bellman-Ford` 算法复杂度为$O(VE)$，`Johnson` 算法会调用V次 `dijkstra`，所以复杂度为$O(VE\lg V)$

2. 实验数据分析

   - 在输入数据中 `input11.txt`，`input31.txt`，`input41.txt`，`input42.txt` 存在负环，在代码中消除负环
   - `input21.txt` 和 `input22.txt` 的输入完全相同
   - 在输出中，无法到达的两点之间的距离用 `NULL` 标记出来

3. 实际复杂度分析



如图所示，从左到右依次为`input11`，`input12`，`input21`,`input22`，`input31`，`input32`，`input41`，`input42`的运行时间，可以看到，对于相同数量级的边的个数，运行时间曲线基本符合$O(VE\lg V)$，对于不同边的个数，对运行时间没有过大的影响

五、实验思考与反思

- 深入理解了和最短路径有关的各种算法：`spfa`，`dijkstra`，`Bellman-Ford`，`Johnson`
- 学习了分析图算法复杂度的相关技巧
- 增强了调试代码的能力