实验报告

杨凌林 PB17000083

实验设备与环境

实验设备

系统版本	Windows 10 专业版
系统版本号	2004
计算机型号	Asus U4100U
处理器	Core i7-8550U

编译环境

IDE	CLion(2020.2.4)
环境	MinGW
编译器	gcc(9.2.0)
CMake	Bundled(3.17.3)

实验内容和要求

实验一 Kruskal 算法

实现求最小生成树的 Kruskal 算法。要求随机生成 N 个顶点的无向图，每一个顶点随机生成 $1 \sim \lfloor N/2 \rfloor$ 条边，其中 $N = 8, 64, 128, 512$，并将生成的随机图写入文件。运行 Kruskal 算法，将求得的最小生成树写入文件，并对运行时长作图，分析其复杂度。

实验二 Johnson 算法

实现求所有点对最短路径的 Johnson 算法。要求随机生成不包含负权值回路的有向图，其中有向图的顶点数 N 的取值分别为 $27, 81, 243, 729$，同一顶点数目对应两种弧的数目 $\lfloor \log 5N \rfloor$，$\lfloor \log 7N \rfloor$。统计算法所需运行时间，并画出时间曲线分析算法性能。

实验方法和步骤

实验一 Kruskal 算法

`src` 文件夹中文件和函数的组织形式如下：

```
main.c
    char *join(char *s1, char *s2)   // 连接字符串
    int main()
graph_generate.h
    struct edge
    struct graph    // 邻接矩阵表示
    int *edge_index_generate(int i, int n)  // 随机生成与当前节点相邻的节点下标
    struct graph graph_generate(int graph_size)
        // 生成规模为 graph_size 的随机无向联通图，连通性已在算法运行过程中保证
    void graph_print(struct graph G, FILE *f)  // 将图 G 输出到文件 f 中
kruskal.h
    struct set_node
    struct set       // 与最小生成树路径有关的两个结构体
    struct set create_set()
    struct set insert_set(struct set A, int i, int j, int a)
    void swap(struct edge *E, int i, int j)
    int max_heapfy(struct edge *E,int n,int i)
    void build_max_heap(struct edge *E,int n)
    void heap_sort(struct edge *E,int n)
    void Union(int *S, int i, int j, int size)
        // 将大小为 size 的数组中与 i, j 同集合的元素合并为一组
    struct set kruscal(struct graph G)
```

几点注意

- 生成随机图时，已经通过生成算法保证所生成的随即图为联通的，方便后续 Kruskal 算法运行。具体生成随机图算法的思路为：依次往当前联通图中插入节点，并添加当前节点和其他节点的边，使加入当前节点后，仍构成一个连通图。重复这个操作，直到节点数达到要求。
- 对集合的并操作，在程序中使用了长度为 `G.size` 的数组 `S` 表示不同节点所在集合。`S` 中数值相同的下标所代表的节点处于同一集合，故每次 `union` 操作复杂度为 $\mathcal{O}(|V|)$。

- 在结果中，由于有一定概率产生权值为 1 的边，而这样的权值是最小的（权值范围为 $[1, 20]$ ），故在最小生成树中，很大一部分都是那些权值为 1 的边。

实验二 Johnson 算法

`src` 文件夹中文件和函数的组织形式如下：

```
main.c
    char *join(char *s1, char *s2)   // 连接字符串
    int main()
graph.generate.h
    struct edge
    struct node
    struct graph   // 邻接表表示
    int *rand_num_generate(int n, int m)   // 生成 n 个互不相同的正整数，范围在 0 - m-1
    struct graph graph_generate(int n, int m, int a, int b)
        // 生成 n 个节点，每个节点 m 条边的无负值环的有向图
        // 边的权值 [a, b]
    void graph_print(struct graph G, FILE *f)  // 将图 G 的信息输出到文件 f 中
bellman_ford.h
    struct source_node
    void init(struct graph G, int s, struct source_node *source) // 初始化
    int relax(struct source_node *source, int i, int j, int w)
    int bellman_ford(struct graph G, int s, struct source_node *source)
        // 若有负环路，返回 0，否则返回 1
dijkstra.h
    struct path_node
    struct source_node_dij
    struct source_node_queue
    struct source_node_queue init_dij(struct graph G, int s) // 初始化
    struct source_node_queue swap_dij
        (struct source_node_queue sourceNodeQueue, int i, int j)
    struct source_node_queue heapfy(struct source_node_queue sourceNodeQueue, int i)
    struct source_node_queue  extract_min
        (struct source_node_queue sourcenodequeue, int *a)
          // 将 min 的下标返回到 a 中
    void Uni_dij(int *S, int u)
    struct source_node_queue decrease_key
        (struct source_node_queue sourceNodeQueue, int i, int key)
    struct source_node_queue relax_dij
        (struct graph G, int u, int v, int w,
         struct source_node_queue sourceNodeQueue)
    struct source_node_queue dijkstra(struct graph G, int s)
johnson.h
    struct graph new_graph(struct graph G)  // 加入新节点 s 后的新图
    int  print_path_i_j
        (struct source_node_queue sourceNodeQueue, int i, int j, FILE *f)
          // 输出 i 到 j 最短路径
    struct source_node_queue *johnson(struct graph G)
```

几点注意

- `dijkstra` 算法针对输入 G_new 实现的，即，添加了新节点 `s` 后的图，并在操作中，忽略了 `s` 的存在，故内部的图大小为 G.size - 1。
- 算法实现时使用的是基于二叉堆的优先队列。

实验结果和分析

实验一 Kruskal 算法

实验截图

$N = 8$ 时生成随机树文件 `input1.txt` 和运行结果文件 `result1.txt` 文件结果截图如下：

input1.txt

```
1  (1, 2, 8)
2  (1, 3, 10)
3  (1, 5, 8)
4  (1, 7, 8)
5  (2, 4, 3)
6  (2, 6, 12)
7  (3, 7, 19)
8  (4, 5, 2)
9  (5, 6, 5)
10 (5, 7, 16)
11 (6, 8, 16)
12 (7, 8, 15)
```

result1.txt

```
1  ( 4, 5, 2)
2  ( 2, 4, 3)
3  ( 5, 6, 5)
4  ( 1, 7, 8)
5  ( 1, 2, 8)
6  ( 1, 3, 10)
7  ( 7, 8, 15)
8  51
```

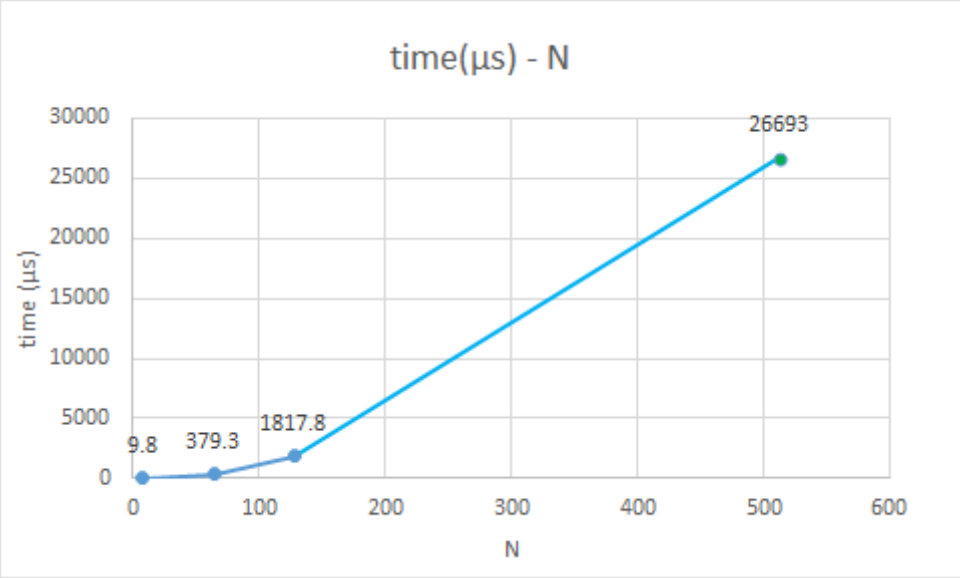
运行时长截图如下：

time.txt

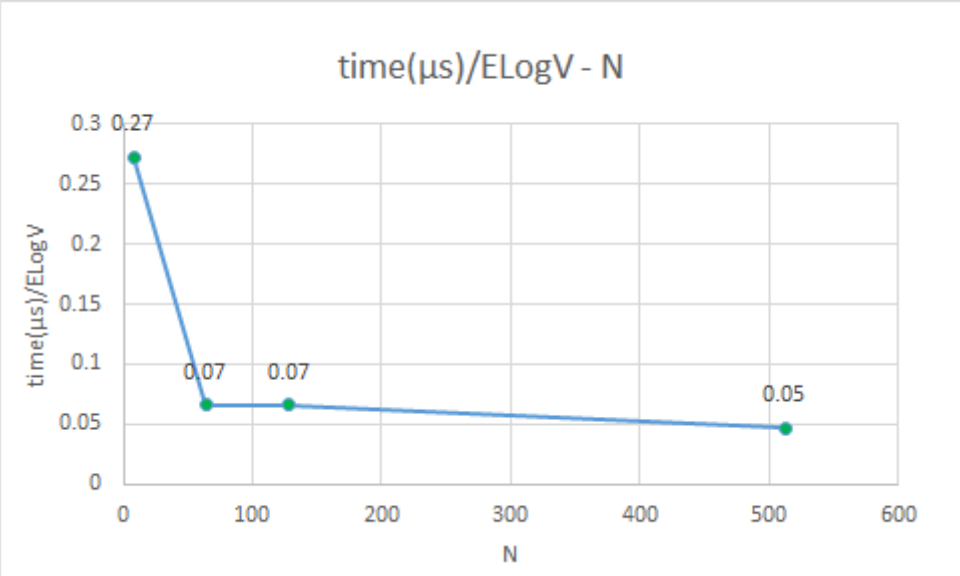
```
9.80 μs
269.30 μs
1927.80 μs
26693.00 μs
```

实验结果分析

画出 `time - N` 图，发现有非线性关系：



为验证课本上 $\mathcal{O}(E\log V)$ 的理论复杂度，我们继续作 `time/ELogV - N` 图，如下所示：



观察图形，可以发现，随着图中节点个数 N 的增大，`time/ElogV` 曲线趋于平缓，除了第一个数据点可以认为是节点数目太小所引起的波动可以忽略以外，算法的渐进常数在 0.05 左右波动。故在实验误差允许范围内，我们可以认为算法的理论复杂度 $\mathcal{O}(E\log V)$ 是正确的。

实验二 Johnson 算法

实验截图

$N = 27$ 且同一顶点出度为 $\log_5 N$ 时随即图文件（部分） `input11.txt` 和算法运行结果文件（部分） `result11.txt` 截图如下：

input11.txt

```
(1, 15, 50)
(1, 9, 19)
(2, 24, 2)
(2, 15, 42)
(3, 17, 8)
(3, 26, 17)
(4, 6, 48)
(4, 23, 30)
(5, 21, 30)
(5, 27, 46)
(6, 27, 41)
(6, 23, 21)
(7, 7, 17)
(7, 25, 36)
(8, 10, 8)
(8, 1, 32)
```

result11.txt

```
(1  0)
(1, 9, 15, 25, 13, 21, 12, 2  162)
(there is no path between 1 and 3)
(1, 9, 15, 6, 27, 4  150)
(there is no path between 1 and 5)
(1, 9, 15, 6  59)
(there is no path between 1 and 7)
(1, 9, 15, 6, 23, 11, 8  162)
(1, 9  19)
(1, 9, 15, 6, 23, 11, 8, 10  170)
(1, 9, 15, 6, 23, 11  125)
(1, 9, 15, 25, 13, 21, 12  115)
(1, 9, 15, 25, 13  95)
(there is no path between 1 and 14)
(1, 9, 15  45)
(1, 9, 15, 25, 13, 16  120)
```

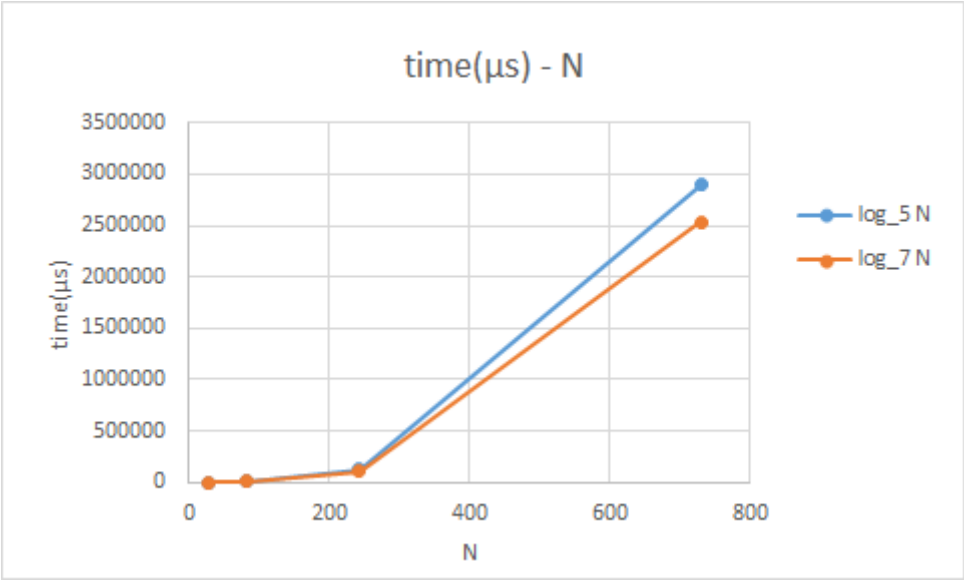
运行时长截图如下：

time.txt

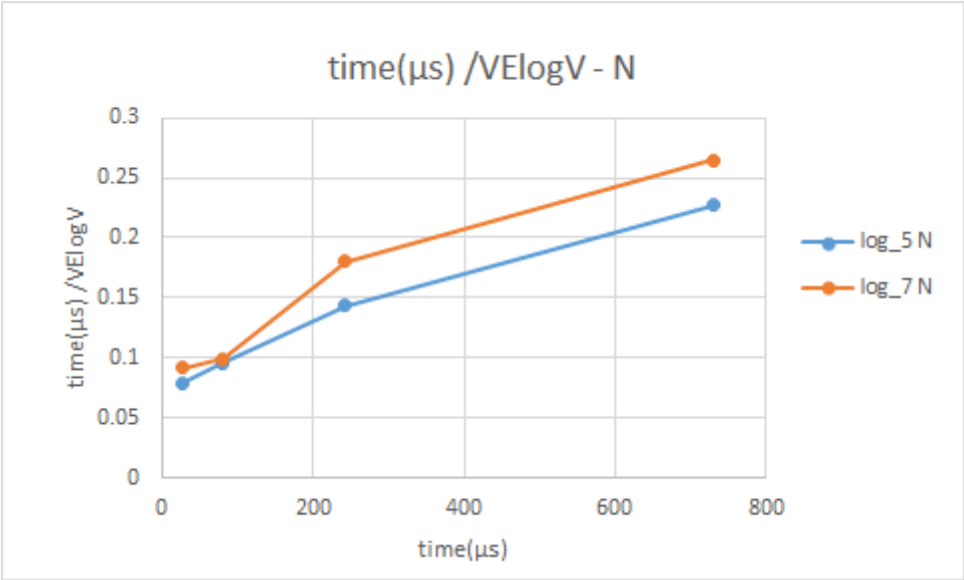
```
time for 27 nodes, log 5 N: 345.40 μs
time for 27 nodes, log 7 N: 209.30 μs
time for 81 nodes, log 5 N: 5030.50 μs
time for 81 nodes, log 7 N: 5110.90 μs
time for 243 nodes, log 5 N: 126654.10 μs
time for 243 nodes, log 7 N: 106169.50 μs
time for 729 nodes, log 5 N: 3120526.60 μs
time for 729 nodes, log 7 N: 2532120.10 μs
```

实验结果分析

对运行时长作图，如下所示：



为了进一步分析算法的复杂度，我们做 `time/VElogV - N` 图，如下所示：



可以看到随着 N 的增大，曲线逐渐变缓，基本上，我们可以判断，使用二叉堆做优先队列时，johnson 算法的复杂度是 $\mathcal{O}(EV \log V)$。

实验总结

通过编程实践，我们实现了几个重要的图算法：kruskal 算法，johnson 算法，bellman-ford 算法，dijkstra 算法（后两个是 johnson 算法中的步骤）。并通过统计真实的运行时长验证了 Kruskal 算法和 johnson 算法的理论复杂度是正确的。

在图数据的处理中，使用不同的算法能得到不同的复杂度，运行时长也各不相同。本次实验中，实现的都是其中较好的一些算法，对以后的学习很有借鉴意义。