实验 4　图论算法

Pb15111604
金泽文

1. 实验要求

- ex1：　实现求有向图的强连通分量的算法。有向图的顶点数 N 的取值分别为：8、16、32、64、128、256，弧的数目为 NlogN,随机生成 NlogN 条弧，统计算法所需运行时间，画出时间曲线。
- ex2：　实现求所有点对最短路径的 Johnson 算法。生成连通的无向图，图的顶点数 N 的取值分别为：8、16、32、64、128、256，边的数目为 NlogN,随机生成 NlogN 条边，边的权重为正数，大小不大于 N，统计算法所需运行时间，画出时间曲线。

2. 实验环境

编译环境：Ubuntu 16.04.3 LTS (WSL- Windows Subsystem for Linux)
Openjdk version "1.8.0_151"
OpenJDK Runtime Environment (build 1.8.0_151)
dot - graphviz version 2.38.0（ex1 画图用）
编程语言：Java SE8
机器内存：16G
时钟主频：2.3GHz

3. Build

为了方便构建，我写了 2 个 Makefile，使用方法如下：

```
Reaper@KZ:/mnt/g/PB15111604-project4/ex1/source$ make help
make                - 只生成SCC.class并且运行
make scc            - 同上
make gen            - 生成随机图存到input中
make clean          - 删除*.class
make help           - 打印以上信息
```

```
Reaper@KZ:/mnt/g/PB15111604-project4/ex2/source$ make help
make                - 只生成Johnson.class并且运行
make johnson        - 同上
make gen            - 生成随机图存到input中,确保连通且无重边
make clean          - 删除*.class
make help           - 打印以上信息
```

实验过程

Ex1：

1. 生成随机图
   针对不同的 N，生成 N 个节点，NlogN 条有向边，不需要考虑权重。
2. 实现 Node 类，field 有表示节点 index 的 key，有表示颜色的 color，表示 finish 时间的 f，表示边信息的链表：

```
public class Node{
    int key;                                    // 图中index
    public COLOR color;                         // 颜色,白灰黑
    public int f;                               // 表示finish的时间
    public Node parent;                         // parent
    public ArrayList<Node> Adj;                 // 邻接表的链表
```

3. 实现 Graph 类，节点存在 vertices 数组中，边信息存在邻接表中，维护 finish 时间的数组 f，最后得到不同强连通分量组成的森林，还要实现 dfs 遍历，加边等 method。

```
class Graph{
    public Node[] vertices;                     // 邻接表
    public int n;                               // 顶点数
    public int[] f;                             // finish数组,方便最后不用排序,直接按降序遍历
    private int time;                           // 全局time
    enum COLOR{WHITE, GREY, BLACK};
    public ArrayList<ArrayList<Node>> forest;
```

4. 实现生成强连通分量算法，要实现图 G 转置的函数 getGT。
5. 处理输入输出信息，尤其是读取图信息，以及生成 dot 文件，再生成 svg 文件的过程。

Ex2：

1. 生成随机图
   针对不同的 N，生成 N 个节点，NlogN 条有向边，边的权重要求都是正的，并且小于 NlogN，这里要考虑不生成重边，要生成连通图。生成连通图的部分通过魔改 ex1 的 SCC 操作，得到强连通分量，判断强连通分量的个数。如果不连通，则重新生成。
2. 实现 VNode 类，以及 ENode 类，后者是为了存储边的权重信息，放在 VNode 类的链表中的，而前者与 ex1 的 Node 有些类似。

```
// 邻接表中表对应的链表的顶点
public class ENode {
    int ivex;       // 该边所指向的顶点的位置
    int weight;     // 该边的权

// 邻接表中表的顶点
public class VNode {
    int key;            // 顶点信息
    int d;              // 到s的距离
    int parent;
    int degree;         // 出度
    ArrayList<ENode> Adj;
```

3. 实现 Graph 类，节点存在 vertices 数组中，边信息保存在邻接表中，需要加边操作。

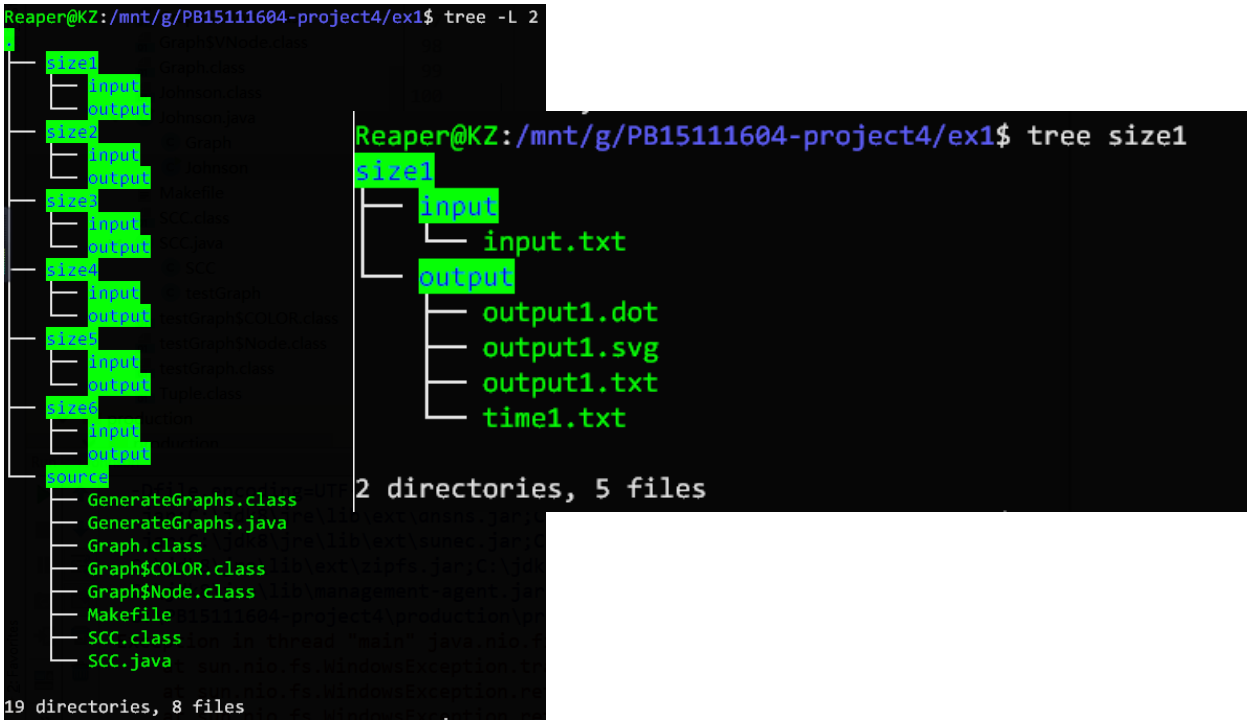
```
class Graph{
    public VNode[] vertices;        // 邻接表
    public int n;                   // 顶点数
```

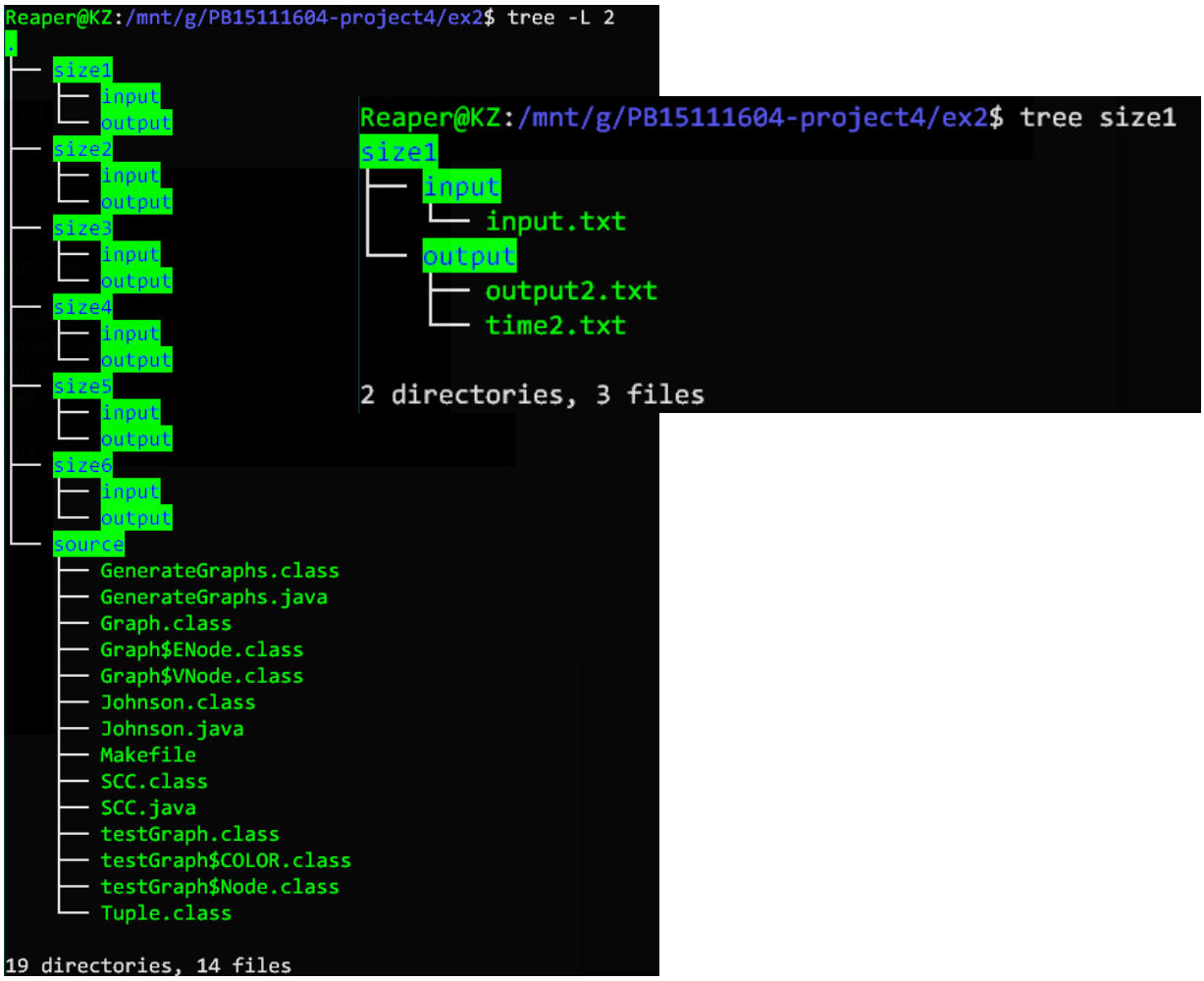
4. 实现 Johnson 算法，由于都是非负边，所以不需要 BellmanFord 调整，只需要粗暴地对每个节点，来一次 dijkstra 操作。Dijkstra 操作所需要的 initializeSingleSource 和 relax 操作都与教材中相似，dijkstra 算法本身也与教材中类似。为了节省时间，这里用的“优先队列”是链表 ArrayList<Integer>。
5. 处理输入输出信息。

最后，制图分析。

4. 实验关键代码截图（结合文字说明）

最后的目录结构如下：





代码结构:
Ex1：

SCC.java 的框架：

```
14  public class SCC {
15      public static boolean CHECKMODE = true;
16      private static boolean ifIntelliJ = true;
17      private static final String ABSPATH = "G:/PB15111604-project4/ex1/";
18      private static final int[] sizes = {1,2,3,4,5,6};
19      private static int nlogn;
20
21      private static int[] originalInts ;
22      private static Graph G, GT;
23
24      private static PrintWriter printTime;
25      private static PrintWriter printOutcome;
26      private static PrintWriter printDot;
27      private static Scanner inputIntegers;
28
29  @   public static void main(String[] args)throws IOException{...}
57
58      // 初始化
59      private static void init(int size)throws IOException{...}
81
82      // 输入
83      public static void input(int n)throws IOException{...}
89
90      // 针对每个size进行处理的主力函数.
91      public static void process(int size){...}
96
97      // 生成图G的转置GT
98      public static Graph getGT(Graph G){...}
108
109     // 生成dot再生成svg.
110     private static void printOutput(int size)throws  IOException{...}
141 }
```

```
147 class Graph{
148     public Node[] vertices;                     // 邻接表
149     public int n;                               // 顶点数
150     public int[] f;                             // finish数组,方便最后不用
        排序,直接按降序遍历
151     private int time;                           // 全局time
152     enum COLOR{WHITE, GREY, BLACK};
153     public ArrayList<ArrayList<Node>> forest;
154
155     // 生成图g的copy
156 @   public Graph(Graph g){...}
165
166     // 生成n个节点的图
167     public Graph(int n){...}
176
177     // 添加边
178     public void addEdge(int x, int y){...}
183
184     // G的深度优先
185     public void dfs(){...}
196
197     // GT的深度优先-按照finish数组的逆序遍历
198     public void dfs(int[] finish){...}
214
215     // G的深度优先递归部分
216 @   private void dfsVist(Node u){...}
229
230     // GT的深度优先递归部分,将访问到的节点添加到scc连通分量里.
231 @   private void dfsVist(Node u, ArrayList<Node> scc){...}
242
243     // 节点类
244     public class Node{...}
256
257 }
```

SCC

Graph::dfs(), dfsVisit(Node u),

Graph::dfs(int[] finish), dfsVisit(Node u, ArrayList<Node> scc)

```
184         // G的深度优先
185         public void dfs(){...}
196
197         // GT的深度优先-按照finish数组的逆序遍历
198         public void dfs(int[] finish){...}
214
215         // G的深度优先递归部分
216     @   private void dfsVist(Node u){...}
229
230         // GT的深度优先递归部分,将访问到的节点添加到scc连通分量里.
231     @   private void dfsVist(Node u, ArrayList<Node> scc){...}
```

这几个 method 都是深度优先遍历所用的.针对 G 和 GT 的不同遍历需求,所以设置了两类.

dfs()(与教材类似):

```
184         // G的深度优先
185         public void dfs(){
186             for(Node node : vertices){
187                 node.color = COLOR.WHITE;
188                 node.parent = null;
189             }
190             time = 0;
191             for(Node node: vertices){
192                 if(node.color == COLOR.WHITE)
193                     dfsVist(node);
194             }
195         }
```

dfsVist(Node u)(与教材类似):

```
215         // G的深度优先递归部分
216     @   private void dfsVist(Node u){
217             u.color = COLOR.GREY;
218             for(Node v : u.Adj){
219                 if(v.color == COLOR.WHITE){
220                     v.parent = u;
221                     dfsVist(v);
222                 }
223             }
224             u.color = COLOR.BLACK;
225             u.f = time;
226             f[time] = u.key;
227             time ++;
228         }
```

其他两个的不同之处,就在于连通分量的额外处理,以及遍历顺序按照 finish 数组的逆序遍历.

Graph::addEdge(int x, int y):

```
177         // 添加边
178         public void addEdge(int x, int y){
179             Node nodeX = vertices[x];
180             Node nodeY = vertices[y];
181             nodeX.Adj.add(nodeY);
182         }
```

SCC::init(), input()

初始化各个参数,并且读入图的信息.

```
56      // 初始化
57      private static void init(int size)throws IOException{
58          nlogn = (1<<(size+2))*(size+2);
59          G = new Graph( n: 1<<(size+2));
60          if(!ifIntelliJ){
61              inputIntegers = new Scanner(Paths.get( first: "../size" + size + "/input/input.txt"), charsetName: "utf-8");
62              input(nlogn);
63              printTime = new PrintWriter( fileName: "../size" + size + "/output/time1.txt", csn: "utf-8");
64              printOutcome = new PrintWriter( fileName: "../size" + size + "/output/output1.txt", csn: "utf-8");
65              printDot = new PrintWriter( fileName: "../size" + size + "/output/output1.dot", csn: "utf-8");
66          }
67          else {
68              inputIntegers = new Scanner(Paths.get( first: ABSPATH + "size" + size + "/input/input.txt"), charsetName: "utf-8")
69              input(nlogn);
70              printTime = new PrintWriter( fileName: ABSPATH + "size" + size + "/output/time1.txt", csn: "utf-8");
71              printOutcome = new PrintWriter( fileName: ABSPATH + "size" + size + "/output/output1.txt", csn: "utf-8");
72              printDot = new PrintWriter( fileName: ABSPATH + "size" + size + "/output/output1.dot", csn: "utf-8");
73          }
74          // 处理原始图G
75          for(int i = 0; i < 2*nlogn; i+=2){
76              G.addEdge(originalInts[i], originalInts[i+1]);
77          }
78      }
79
80      // 输入
81      public static void input(int n)throws IOException{
82          originalInts  = new int[2*n];
83          for(int i = 0; i < 2*n; i++){
84              originalInts[i] = inputIntegers.nextInt();
85          }
86      }
```

SCC::getGT(Graph G)

生成图 G 的转置图,操作很简单,边按照逆序存入.

```
95          // 生成图G的转置GT
96          public static Graph getGT(Graph G){
97              // O(E)
98              Graph GT = new Graph(G);
99              for(int i= 0; i < G.n; i++){
100                 for(Graph.Node v: G.vertices[i].Adj){
101                     GT.addEdge(v.key, i);
102                 }
103             }
104             return GT;
105         }
```

SCC::printOutput(int size)

针对图 GT 最后的信息,生成 dot 格式,再通过 dot 命令得到 svg 图片,有向图的信息按照一开始的 input.txt 得到的 G 画出,针对不同的强连通分量,通过不同颜色显示节点.但是由于边比较多,所以强连通分量比较少.
另外,这里因为 size 为 5 和 6 时数据量太大,生成时间太长,显示效果不好,所以只生成了 5 和 6 的 dot 文件

```
107     // 生成dot再生成svg.
108     private static void printOutput(int size)throws  IOException{
109         String[] colors = {"red", "black", "orange", "yellow", "green", "blue", "purple", "grey", "white"};
110         int i = 0;
111         // dot : digraph
112         printDot.printf("digraph {\n");
113         printDot.flush();
114         for(ArrayList<Graph.Node> tree: GT.forest){
115             printOutcome.printf("("); printOutcome.flush();
116             System.out.printf("(");
117             String color = colors[i++];
118             for(Graph.Node node: tree){
119                 printOutcome.printf("%5d", node.key);
120                 printOutcome.flush();
121                 System.out.printf("%5d", node.key);
122                 // dot : color
123                 printDot.printf("    %d [style=filled color=\"%s\"];\n", node.key, color); printDot.flush();
124             }
125             printOutcome.printf(")\n"); printOutcome.flush();
126             System.out.printf(")\n");
127         }
128         // dot : edge
129         int n = originalInts.length;
130         for(i = 0; i < n; i+=2){
131             printDot.printf( "    %d -> %d;\n", originalInts[i], originalInts[i+1] ); printDot.flush();
132         }
133         printDot.printf("}"); printDot.flush();
134         // 对于size>4,生成svg要很久,所以这个可以当做选择项,如果要生成,请去掉if
135         if(size < 5)
136             Runtime.getRuntime().exec( command: "dot -Tsvg ../size" + size + "/output/output1.dot -o ../size" + size +
137                 "/output/output1.svg");
138     }
139 }
```

Ex2:

Johnson.java 代码框架:

```
17  public class Johnson {
18      public static boolean CHECKMODE = true;
19      private static boolean ifIntelliJ = true;
20      private static final String ABSPATH = "G:/PB15111604-project3/ex2/";
21      private static final int[] sizes = {1,2,3,4,5,6};
22      private static int nlogn;
23      public static final int MAX_NUM = 10000000;
24
25      private static int[] originalInts ;
26      private static Graph G;
27      private static long start, endurance;
28
29      private static PrintWriter printTime;
30      private static PrintWriter printOutcome;
31      private static Scanner inputIntegers;
32
33
34  @   public static void main(String[] args)throws IOException{...}
59
60      // 初始化
61      private static void init(int size)throws IOException{...}
81
82      // 输入
83      public static void input(int n){...}
89
90      // 输出最短路径
91      private static void printOutput(String path)throws  IOException{...}
95
96      // 针对每个size进行处理的主力函数.
97      public static void process(int size)throws IOException{...}
112
113     // 初始化源节点,同教材
114     private static void initializeSingleSource(int s){...}
```

```
112     // 初始化源节点,同教材
113     private static void initializeSingleSource(int s){...}
121
122     // 松弛操作,同教材
123     private static void relax(int u, int v, int w){...}
129
130     // 得到最短路径字符串
131 @   private static String shortestPath(int v){...}
137
138     // dijkstra算法, int s 为源节点
139     private static void dijkstra(int s){...}
165
166 }
167
168 class Graph{
169     public VNode[] vertices;      // 邻接表
170     public int n;                 // 顶点数
171
172     // 生成n个节点的图
173     public Graph(int n){...}
180
181     // 添加边
182     public void addEdge(int x, int y, int w){...}
190
191     // 邻接表中表对应的链表的顶点
192     public class ENode {...}
201
202     // 邻接表中表的顶点
203     public class VNode {...};
215 }
```

Graph::ENode,VNode 类

ENode 类是为了存储边的权重信息，放在 VNode 类的链表中的，
VNode 类与 ex1 的 Node 有些类似

```
191         // 邻接表中表对应的链表的顶点
192         public class ENode {
193             int ivex;       // 该边所指向的顶点的位置
194             int weight;     // 该边的权
195 
196             public ENode(int key, int w){
197                 this.ivex = key;
198                 this.weight = w;
199             }
200         }
201 
202         // 邻接表中表的顶点
203         public class VNode {
204             int key;            // 顶点信息
205             int d;              // 到s的距离
206             int parent;
207             int degree;         // 出度
208             ArrayList<ENode> Adj;
209 
210             public VNode(int key){
211                 this.key = key;
212                 this.Adj = new ArrayList<>();
213             }
214         }
```

Johnson::shortestPath(int v)

得到到 v 的最短路径,用字符串表示

```
130         // 得到最短路径字符串
131     @   private static String shortestPath(int v){
132             if(v == -1)
133                 return "";
134         ⏲   return shortestPath(G.vertices[v].parent) + " " + v;
135 
136         }
```

Johnson:: initializeSingleSource(int s), relax(int u, int v, int w)

这两个函数和教材中一致

```
112         // 初始化源节点,同教材
113         private static void initializeSingleSource(int s){
114             int n = G.n;
115             for(int i = 0; i < n; i++){
116                 G.vertices[i].d = MAX_NUM;
117                 G.vertices[i].parent = -1;
118             }
119             G.vertices[s].d = 0;
120         }
121 
122         // 松弛操作,同教材
123         private static void relax(int u, int v, int w){
124             if(G.vertices[v].d > G.vertices[u].d + w){
125                 G.vertices[v].d = G.vertices[u].d + w;
126                 G.vertices[v].parent = u;
127             }
128         }
```

Johnson::dijkstra(int s)

与教材中代码类似,队列用链表 ArrayList<Integer>实现,每次 extract-min 都取链表中最小值.

```
130         // 得到最短路径字符串
131     @   private static String shortestPath(int v){...}
137 
138         // dijkstra算法, int s 为源节点
139         private static void dijkstra(int s){
140             int n = G.n;
141             Integer u = 0, min;
142 
143             ArrayList<Integer> q = new ArrayList<>();
144             for(int i = 0; i < n; i++)
145                 q.add(new Integer(i));
146 
147             initializeSingleSource(s);
148             while(n > 0){
149                 // extract-min
150                 min = MAX_NUM;
151                 for(Integer i : q)
152                     if(G.vertices[i].d < min){
153                         u = i;
154                         min = G.vertices[i].d;
155                     }
156                 // u = minimum, min = minimun
157                 q.remove(u);
158                 n--;
159 
160                 for(Graph.ENode v: G.vertices[u].Adj){
161                     relax(u, v.ivex, v.weight);
162                 }
163             }
164         }
```

Johnson::process(int size)
负责 Johnson 的主体部分,并且负责输出最短路径信息到文件中.

```
95      // 针对每个size进行处理的主力函数.
96      public static void process(int size)throws IOException{
97          for(int i = 0; i < G.n; i ++){
98              dijkstra(i);
99              String path;
100             for(int j = 0; j < G.n; j++){
101                 path = i + "->" + j + " (";
102                 path = path + shortestPath(j);
103                 path = path + ")" + " length: " + G.vertices[j].d;
104                 endurance += System.nanoTime() - start;
105                 printOutput(path);
106                 System.out.println(path);
107                 start = System.nanoTime();
108             }
109         }
110     }
```

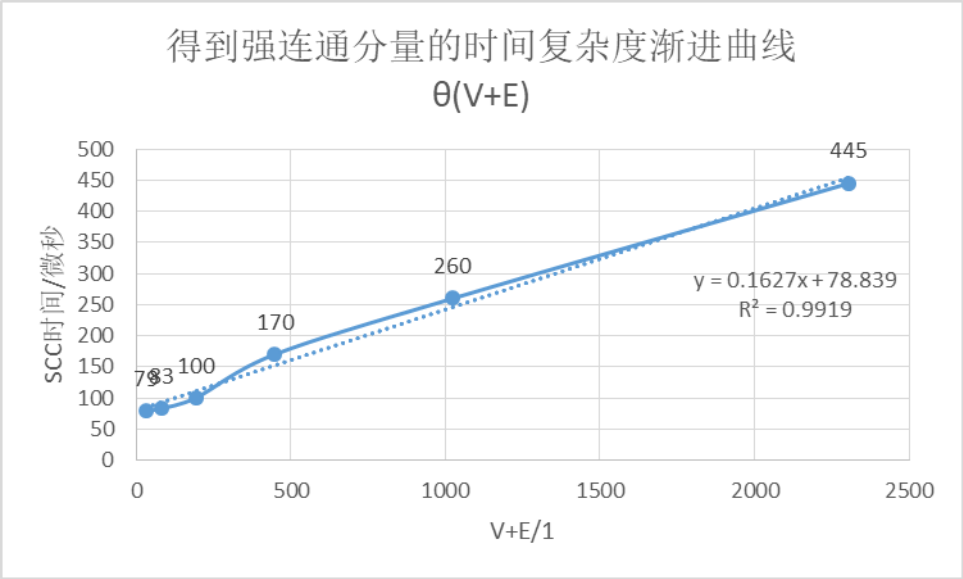
5. 实验结果、分析（结合相关数据图表分析）

//注意:
//　在实验过程中为了减少硬件 cache 策略对分析的影响,所以在给定的 size 之外,
//　我首先多跑了一个 8,来避免对后面 16,32,64,128,256 的影响.

这是 ex1 得到的数据

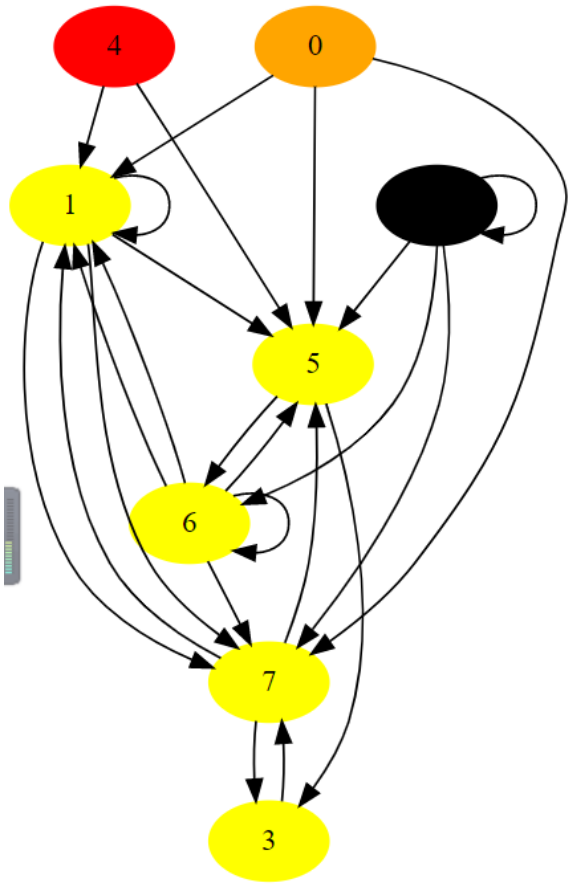
```
Reaper@KZ:/mnt/g/PB15111604-project4/ex1/source$ make
   79000 nanoseconds.
   83000 nanoseconds.
  100000 nanoseconds.
  170000 nanoseconds.
  260000 nanoseconds.
  445000 nanoseconds.
```

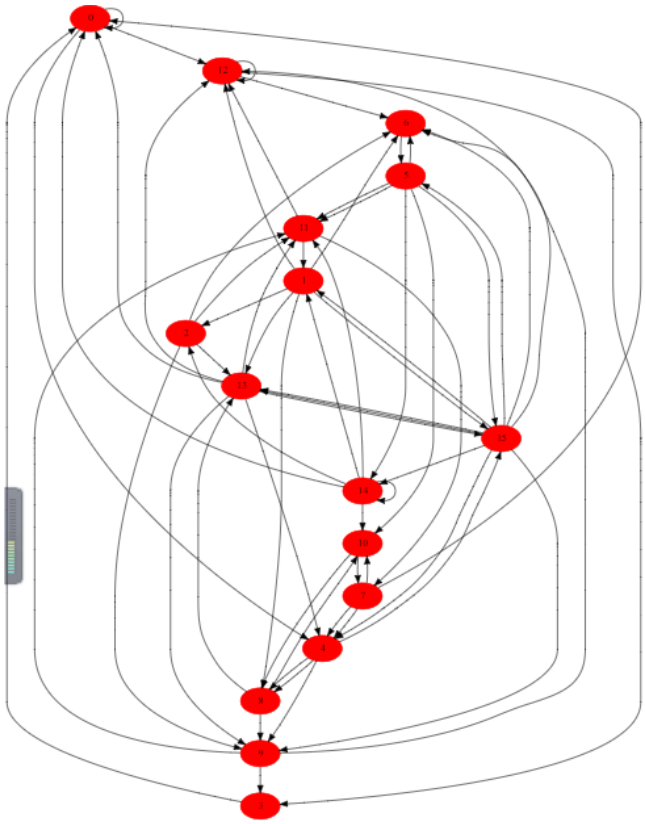
V	8	16	32	64	128	256
E	24	64	160	384	896	2048
SCC 时间/微秒	79	83	100	170	260	445
V+E	32	80	192	448	1024	2304



十分符合教材中提到的 θ(V+E)的渐进时间复杂度!并且 R^2 有 0.9919,拟合很好!

附两张 graphviz 做的图，这两张是用 size1 和 size2 的 input.txt 做的图,

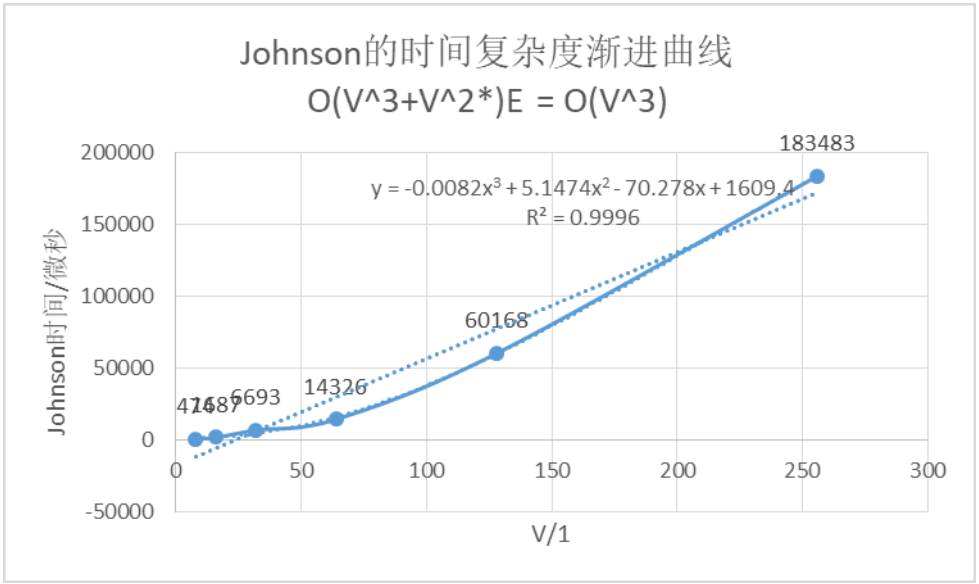




这是 ex2 得到的数据

```
Reaper@KZ:/mnt/g/PB15111604-project4/ex2/source$ make
   474000 nanoseconds.
  1687000 nanoseconds.
  6693000 nanoseconds.
 14326000 nanoseconds.
 60168000 nanoseconds.
183483000 nanoseconds.
```

V	8	16	32	64	128	256
V^3	512	4096	32768	262144	2097152	16777216
Johnson 时间/微秒	474	1687	6693	14326	60168	183483



由于优先队列使用的是链表,所以一次 dijkstra 的渐进时间复杂度 O(V^2+E),所以 Johnson 整体的时间复杂度为 O((V^2+E)*V) = O(V^3)

由上图可知,三次方拟合的效果非常好,R^2 有 0.9996,符合 O(V^3)的渐进时间复杂度.

6. 实验心得

a) 首先，通过本次实验，加深了强连通分量和 Johnson 算法,以及 Dijkstra 孙发的理解，这是最重要的。尤其是通过切身地动手实现图的节点以及各种函数，发现了很多以前没有考虑到的细节。这是最大的收获。
b) 其次，通过本次实验，发现了图论算法没有想象中那么难处理!克服了之前对图论的心理障碍。
c) 再其次，通过本次实验，我学习了 graphviz 的使用，知道了生成有向图，生成节点，生成 dot，svg，png 等的方法。这是我第一次学会用程序做图！一大飞跃！
d) 最后，和上一次还有上上一次还有上上上一次一样，感谢可爱哒助教读到这里，感谢的同时心疼一下。。。
e) 算法很美很重要，要加深理解与思考！
f) 祝好！

g) 最后一次实验了,就这么结束了,嘤嘤嘤.
h) 么么哒！