

1 问题分析

1.1 第一部分

1.1.1 问题1：建图

本题中代码已经给出。

1.1.2 问题2：判断图是否连通

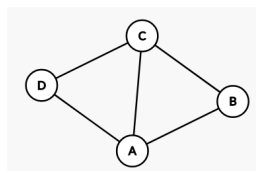
从一个点开始遍历，如果能遍历到所有的点，则图连通。

1.1.3 计算图中每个点的度

节点度是指和该节点相关联的边的条数，对于有向边是入度加出度，对无向图则是度。

1.1.4 计算图的聚类系数

点的聚类系数是所有与它相连的顶点之间所连的边的数量，除以这些顶点之间可以连出的最大边数。图的聚类系数是所有点的聚类系数的均值。



举例，A的邻居为B、C、D，B、C、D之间的边有2条而B、C、D三个点之间可以连出的最大边数是3（两两相连），所以A的聚类系数是 $\frac{2}{3}$ ；B有两个邻居，它们正好相连，所以B的聚类系数是1；同理，C的聚类系数是1；D的聚类系数是 $\frac{2}{3}$ 。综上所述，这个图的聚类系数是 $\frac{5}{6}$ 。

1.1.5 若图连通，使用Dijkstra算法计算单源最短路径

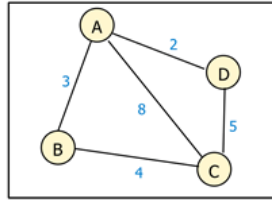
1.1.6 若图连通，计算图的直径、半径

节点距离：指的是两个节点间的最短路径的长度。

Eccentricity：这个参数描述的是从任意一个节点，到达其他节点的最大距离

Diameter：图中的最大的**Eccentricity**

Radius：图中的最小的**Eccentricity**



举例, $\text{Eccentricity}(A) = \text{ABC} = 7$;

$\text{Eccentricity}(B) = \text{BAD} = 5$;

$\text{Eccentricity}(C) = \text{CBA} = 7$;

$\text{Eccentricity}(D) = \text{DC} = \text{DAB} = 5$;

所以半径是5, 直径是7。

1.2 第二部分

我们提供了深圳地铁的线路图, 请同学们自行读取文件建图 (文件格式在ppt中说明), 并回答以下几个问题:

1.2.1 这个图是连通的吗?

同第一部分一样, 这里可以采用从一个点开始遍历, 判断所有的点是否可达来判断图是否连通。

1.2.2 线路图中换乘线路最多的站点是哪个? 共有几条线路通过?

这实际上是求最大度的节点以及最大度的问题。

1.2.3 该线路图的直径和半径是多少?

算法同第一部分。

1.2.4 从大学城站到机场站最少需要多少时间? 请打印最短路径上的站点名称

这是Dijkstra算法在寻找最短路径上的应用。

2 详细设计

2.1 设计思想

2.1.1 第一部分

2.1.1.1 问题1: 建图

本题中代码已经给出。

2.1.1.2 问题2：判断图是否连通

从一个点开始遍历，如果能遍历到所有的点，则图连通。判断图是否连通也有其他方法，比如并查集法：依次判断每一条边是否可以把每一个连通块合成一个连通块。考虑到代码复用性，这里采用遍历法。

2.1.1.3 计算图中每个点的度

节点度是指和该节点相关联的边的条数，对于有向边是入度加出度，对无向图则是度。在本题中是无向图，故依次计算每一个点可以向外引出多少条路径即为这个点的度。记录最大度和对应节点。

2.1.1.4 计算图的聚类系数

点的聚类系数是所有与它相连的顶点之间所连的边的数量，除以这些顶点之间可以连出的最大边数。图的聚类系数是所有点的聚类系数的均值。

只要计算出上面一段对应的数即可。注意可能舍去一些节点的值。

聚类系数 = $\frac{n}{C_k^2} = \frac{n}{\frac{k(k-1)}{2}}$ ，其中 n 为边数， k 为顶点的度。

2.1.1.5 若图连通，使用Dijkstra算法计算单源最短路径

见代码。

2.1.1.6 若图连通，计算图的直径、半径

见代码。

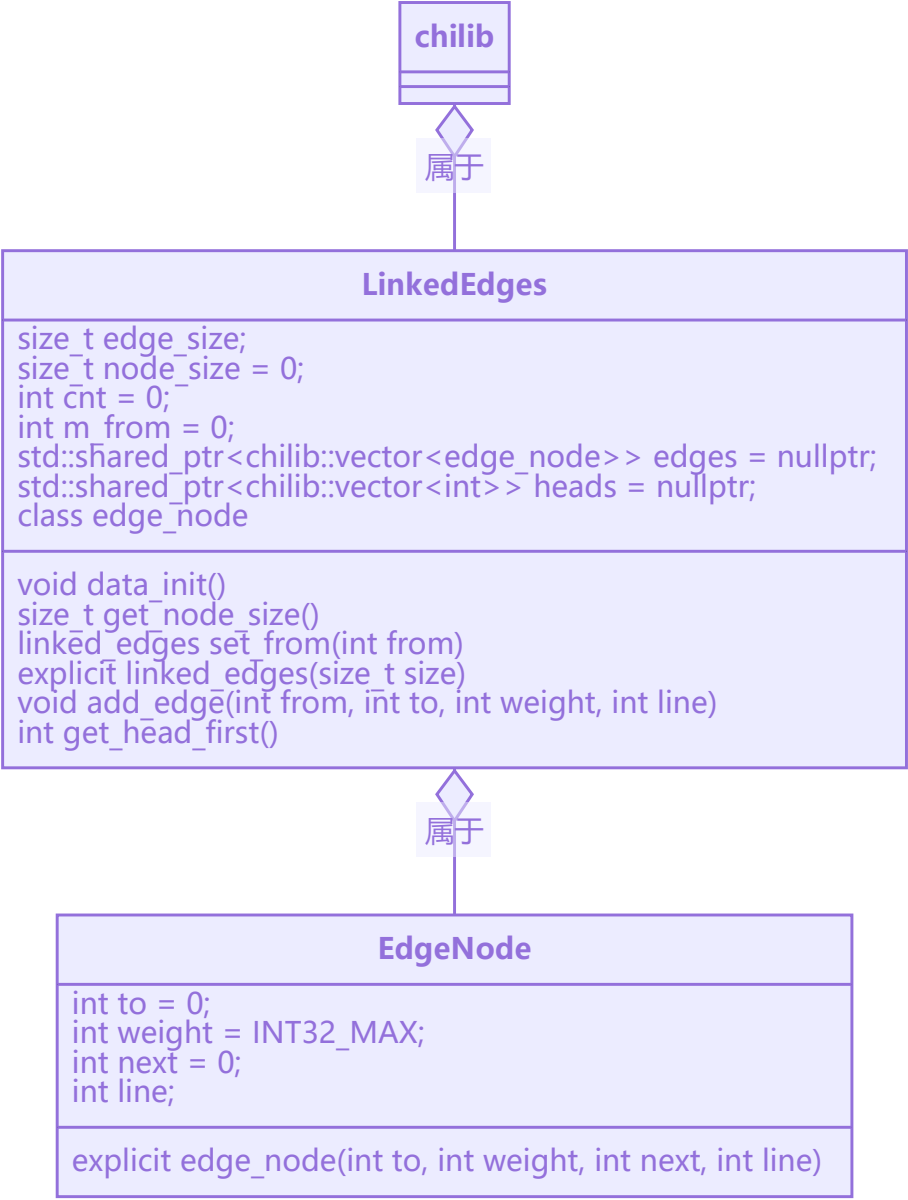
2.2 储存结构和操作

2.2.1 储存结构

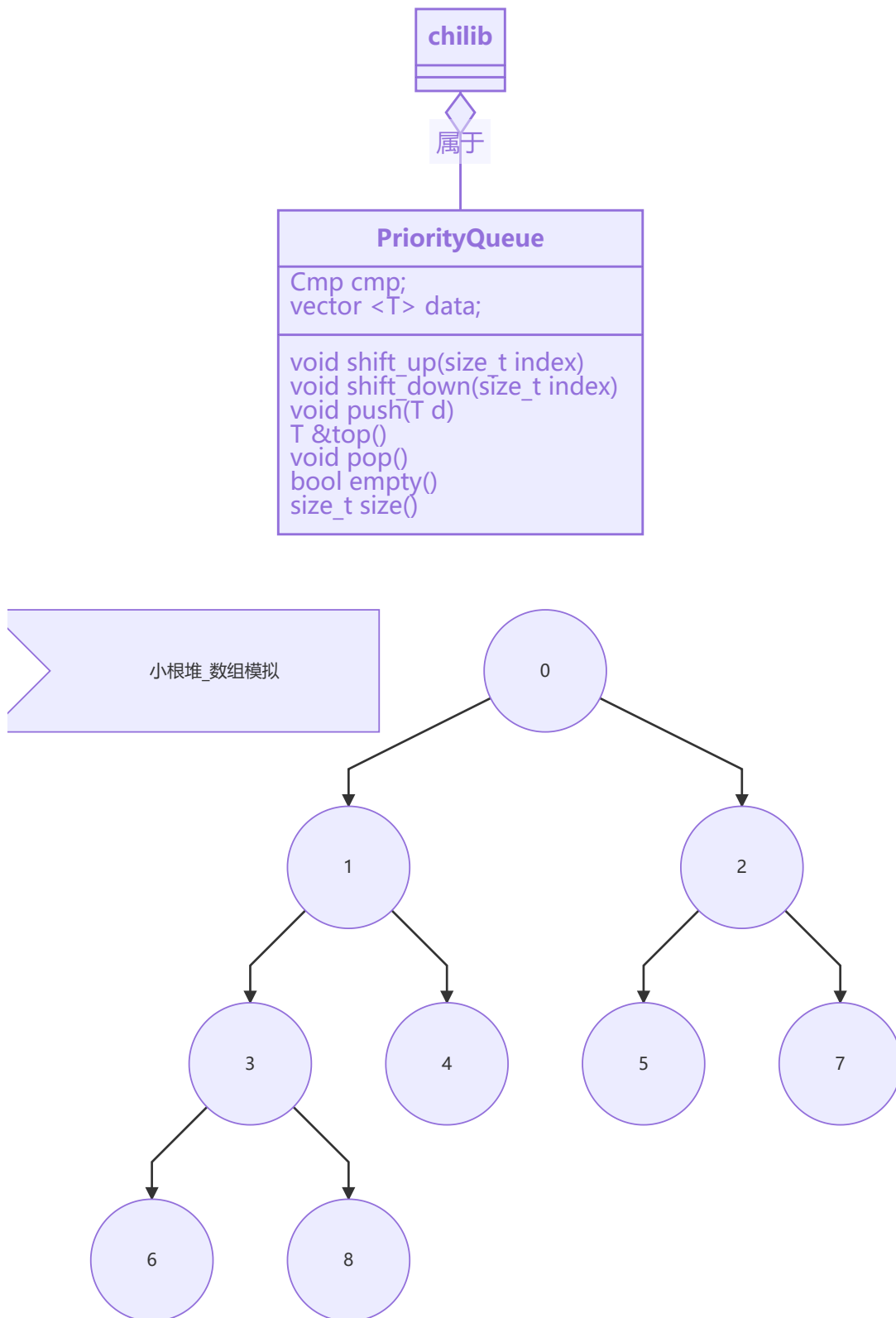
第一部分

```
1 // 邻接表储存结构
2 typedef char vextype[20];
3 typedef struct {
4     int N, E;          // N是顶点数，E是边数
5     int **matrix;      // 储存邻接矩阵
6     vextype *vertex;   // 存储节点的名字
7 } Graph;
```

第二部分：链式前向星储存结构



第二部分：优先队列储存结构



2.2.2 涉及的操作

优先队列中:

```
1  /*!
2   * 上浮操作
3   * @param index 操作节点
4   */
5  void shift_up(size_t index) {
6      if (cmp(data[father(index)], data[index])) {
7          std::swap(data[index], data[father(index)]);
8          shift_up(father(index));
9      }
10 }
11
12 /*!
13 * 下沉操作
14 * @param index 操作节点
15 */
16 void shift_down(size_t index) {
17     if (left(index) >= data.length()) return;
18     if (right(index) >= data.length()) {
19         if (cmp(data[index], data[left(index)])) {
20             std::swap(data[left(index)], data[index]);
21             shift_down(left(index));
22         }
23         return;
24     }
25     if (cmp(data[right(index)], data[left(index)])) {
26         if (cmp(data[index], data[left(index)])) {
27             std::swap(data[left(index)], data[index]);
28             shift_down(left(index));
29         }
30     } else {
31         if (cmp(data[index], data[right(index)])) {
32             std::swap(data[right(index)], data[index]);
33             shift_down(right(index));
34         }
35     }
36 }
37
```

```

38  /*!
39   * 向队列尾部添加元素
40   * @param d 元素
41   */
42  void push(T d) {
43      data.emplace_back(d);
44      shift_up(data.length() - 1);
45  }
46
47  /*!
48   * 取队列头元素
49   * @return 元素引用
50   */
51  T &top() {
52      empty_check();
53      return data[0];
54  }
55
56  /*!
57   * 弹出队列头元素
58   */
59  void pop() {
60      empty_check();
61      T back = data[data.length() - 1];
62      data.pop_back();
63      if (data.empty()) return;
64      data[0] = back;
65      shift_down(0);
66  }

```

链式前向星中：

```

1  /*!
2   * 设置容器遍历的边起点节点
3   * @param from 起点节点
4   * @return 已经改变之后的链式前向星对象引用
5   */
6  linked_edges &set_from(int from) {
7      // 并不线程安全呢

```

```

8         m_from = from;
9         return *(this);
10    }
11
12    /*!
13     * 加单向边
14     * @param from
15     * @param to
16     * @param weight
17     * @param line 附加: 第几条线
18     */
19    void add_edge(int from, int to, int weight, int line) {
20        cnt++;
21        (*edges)[cnt] = edge_node(to, weight, (*heads)[from], line);
22        (*heads)[from] = cnt;
23    }
24
25    /*!
26     * 取得第一个有边的节点
27     * @return
28     */
29    int get_head_first() {
30        int head_first = 0;
31        while ((*heads)[head_first] == 0) head_first++;
32        return head_first;
33    }

```

地铁图操作:

```

1    // 判断图是否联通
2    bool is_map_connected(chilib::linked_edges &edges) {
3        chilib::vector<bool> visited(edges.get_node_size() + 1);
4        int head_first = edges.get_head_first();
5        // BFS
6        chilib::vector<int> queue;
7        queue.emplace_back(head_first);
8        visited[head_first] = true;
9        while (!queue.empty()) {
10            int from = queue.pop_front();

```



```

11     try {
12         for (const auto &edge : edges.set_from(from)) {
13             int to = edge.to;
14             if (!visited[to]) {
15                 visited[to] = true;
16                 queue.emplace_back(to);
17             }
18         }
19     } catch (std::out_of_range &e) {
20         std::cerr << e.what() << " when from = " << from << std::endl;
21         throw e;
22     }
23 }
24 for (size_t i = 1; i <= edges.get_node_size(); i++)
25     if (!visited[i]) {
26 //         printf("ID %d not visited!\n", i);
27         return false;
28     }
29 return true;
30 }
31
32 // 找经过最多线路的站点
33 int find_most_exchanged(chilib::linked_edges &edges, int &max_degree)
34 {
35     int id = -1;
36     for (size_t i = 1; i <= edges.get_node_size(); i++) {
37         // degree 即为当前节点度
38         int degree = 0;
39         auto it = edges.set_from(i).begin();
40         while (it != edges.end()) {
41             ++it;
42             degree++;
43         }
44         if (degree > max_degree) {
45             // 取节点度最大值并且记录节点id
46             max_degree = degree;
47             id = i;
48         }
49     }
50 }

```

```

48     }
49     return id;
50 }
51
52 void dijkstra(int start, chilib::linked_edges &edges,
53              chilib::vector<int> &distance,
54              chilib::vector<int> *path) {
55     // 初始化 visited, path 和 distance 数组
56     chilib::vector<bool> visited(edges.get_node_size() + 1);
57     distance = chilib::vector<int>(edges.get_node_size() + 1);
58     if (path) *path = chilib::vector<int>(edges.get_node_size() + 1);
59     // 距离初始化为最大值
60     for (auto &d : distance) d = METRO_LEN_MAX;
61     // 虽然这两个优先队列用法一致，但是自己写的这个得开 -O2 性能才赶得上STL的...
62     // std::priority_queue<node_order> q;
63     chilib::priority_queue<node_order> q;
64     // 记录起点距离
65     distance[start] = 0;
66     // 添加起点
67     q.push(node_order(start, 0));
68     while (!q.empty()) {
69         // 取出当前可访问到的边的最小的边
70         node_order top = q.top();
71         q.pop();
72         if (visited[top.pos]) continue;
73         int from = top.pos;
74         visited[from] = true;
75
76         // 利用链式前向星容器遍历边
77         for (const auto &edge : edges.set_from(from)) {
78             int to = edge.to, weight = edge.weight;
79             // 更新最短边数据
80             if (distance[to] > distance[from] + weight) {
81                 distance[to] = distance[from] + weight;
82                 // 添加路径
83                 if (path) (*path)[to] = from;
84                 if (!visited[to]) {
85                     q.push(node_order(to, distance[to]));

```

```

86         }
87     }
88 }
89 }
90 }
91
92 // 计算ecc、直径、半径
93 void compute_ecc(chilib::linked_edges &edges, int &d, int &r) {
94     d = 0, r = METRO_LEN_MAX;
95     for (size_t i = 1; i <= edges.get_node_size(); i++) {
96         int ecc = 0;
97         chilib::vector<int> distance;
98         dijkstra(i, edges, distance, nullptr);
99         // ecc: 单源距离最大值
100         for (size_t j = 1; j <= edges.get_node_size(); j++) {
101             ecc = ecc > distance[j] ? ecc : distance[j];
102         }
103         // d: ecc 的最大值, r: ecc 的最小值
104         d = d < ecc ? ecc : d;
105         r = r > ecc ? ecc : r;
106     }
107 }
108
109 // 主函数
110 int main() {
111     // 配置
112     // 打开这个开关, 会使用Graphviz绘制整个地铁站的图像
113     const bool draw_image = false;
114     // 起点和目的地名称
115     chilib::string station1_name = "大学城",
116         station2_name = "机场";
117     // station2_name = "深圳湾公园";
118     // 文件名
119     const char *metro_data = "metro.txt", *metro_name_data =
120         "no2metro.txt";
121     chilib::linked_edges edges(METRO_EDGES_MAX);
122     chilib::vector<chilib::string> names;
123     data_read_metro(metro_data, edges);

```

```

123     data_read_metro_names(metro_name_data, names);
124     printf("这个图%s联通的.\n", is_map_connected(edges) ? "是" : "否");
125     int max_degree = 0;
126     int most_exchanged = find_most_exchanged(edges, max_degree);
127     printf("线路图中换乘线路最多的站点是%s, 共有 %d 条线路通过.\n",
names[most_exchanged - 1].c_str(), max_degree);
128     chilib::vector<int> distance;
129     dijkstra(most_exchanged, edges, distance, nullptr);
130     // 打印最短长度信息
131     // printf("distance: ");
132     // for (const auto d : distance) {
133     //     printf("%d ", d == METRO_LEN_MAX ? -1 : d);
134     // }
135     // puts("");
136     int d, r;
137     compute_ecc(edges, d, r);
138     printf("该线路图的直径是 %d, 半径是 %d.\n", d, r);
139     int station1 = -1, station2 = -1;
140     for (size_t n = 1; n < names.size(); n++) {
141         if (names[n] == station1_name) station1 = n;
142         if (names[n] == station2_name) station2 = n;
143     }
144     if (station1 < 0 || station2 < 0) {
145         printf("找不到站点: %s 或者 %s!\n", station1_name.c_str(),
station2_name.c_str());
146         return 1;
147     }
148     // printf("station1: %d, %s; station2: %d, %s\n", station1,
station1_name.c_str(), station2, station2_name.c_str());
149     station1++, station2++;
150     chilib::vector<int> path;
151     dijkstra(station1, edges, distance, &path);
152     printf("从%s站到%s站最少需要 %d 分钟, 路径: \n", station1_name.c_str(),
station2_name.c_str(), distance[station2]);
153     // 用栈把路径反过来
154     chilib::vector<int> stack;
155     int p = station2;
156     while (p > 0) {

```

```

157     stack.emplace_front(p - 1);
158     if (p == station1) break;
159     p = path[p];
160 }
161 // 打印路径
162 for (const auto st : stack) {
163     printf("%s", names[st].c_str());
164     if (st != station2 - 1) printf(" --> ");
165 }
166 puts("");
167 if (draw_image) draw(edges, names);
168 return 0;
169 }

```

3 用户手册

本项目中使用了两部分自己写的库：`linked_edges` 和 `priority_queue`，使用说明如下：

3.1 使用说明 - `chilib::priority_queue`

3.1.1 使用

1. 包含文件：`queue.hpp`
2. `chilib::priority_queue<int, chilib::greater<int>> q;`
3. `q.push(1);`
4. `q.pop();`

详细：

```

1 //
2 // Created by Chiro on 2021/5/1.
3 //
4
5 #include <cstdio>
6 #include <cstdlib>
7 #include "chilib/queue.hpp"
8
9 int main() {
10     chilib::priority_queue<int, chilib::less<int>> q;
11     for (int i = 1000; i >= 0; i--) {

```

```

12     int t = rand() % 1000;
13     q.push(t);
14     printf("pushed: %d\n", i);
15 }
16 while (!qs.empty()) {
17     printf("%d\n", q.top());
18     q.pop();
19 }
20 return 0;
21 }

```

3.1.2 接口说明

大致和 `STL` 库一致，区别在性能比 `STL` 低、只会使用 `chilib::vector` 作容器。

```

1  /*!
2   * 向队列尾部添加元素
3   * @param d 元素
4   */
5  void push(T d);
6  /*!
7   * 取队列头元素
8   * @return 元素引用
9   */
10 T &top();
11 /*!
12 * 弹出队列头元素
13 */
14 void pop();
15 /*!
16 * 判断队列是否为空
17 * @return 是否为空
18 */
19 bool empty();
20 /*!
21 * 取得队列长度
22 * @return 队列长度
23 */
24 size_t size();

```

3.2 使用说明 - chilib::linked_edges

3.2.1 使用说明

包含文件: `linked_edges.hpp`。

示例:

```
1  #include <cstdio>
2  #include "chilib/linked_edges.hpp"
3
4  int main() {
5      // 最大节点数
6      const int EDGES_MAX = 3;
7      chilib::linked_edges edges(EDGES_MAX);
8      // 添加边
9      edges.add_edge(1, 2, 1);
10     edges.add_edge(2, 1, 1);
11     // 遍历边
12     for (int from = 1; from < EDGES_MAX; from++)
13         for (const auto &edge: edges.set_from(from))
14             printf("%d ==> %d\n", from, edge.weight, edge.to);
15     return 0;
16 }
```

3.2.2 接口说明

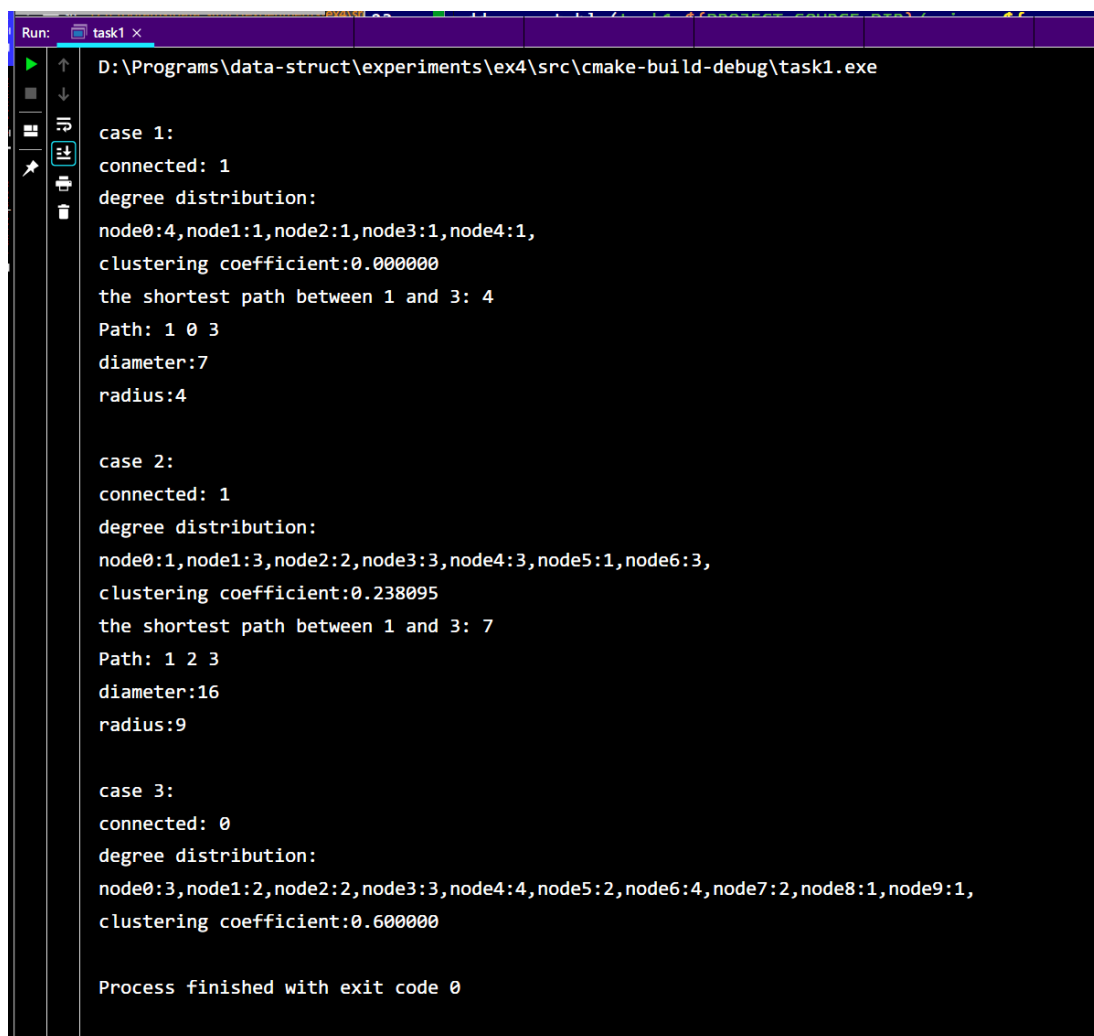
```
1  /*!
2   * 构造函数
3   * @param size 最大节点编号
4   */
5  explicit linked_edges(size_t size);
6  /*!
7   * 设置容器遍历的边起点节点
8   * @param from 起点节点
9   * @return 已经改变之后的链式前向星对象引用
10 */
11 linked_edges &set_from(int from);
12 /*!
13 * 加单向边
14 * @param from
```

```

15     * @param to
16     * @param weight
17     */
18 void add_edge(int from, int to, int weight);
19 /*!
20  * 取得第一个有边的节点
21  * @return
22  */
23 int get_head_first();

```

4 运行结果



```

Run: task1 x
D:\Programs\data-struct\experiments\ex4\src\cmake-build-debug\task1.exe

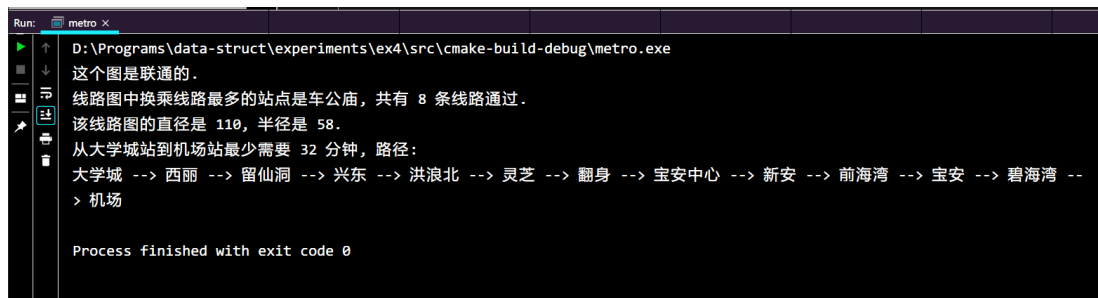
case 1:
connected: 1
degree distribution:
node0:4,node1:1,node2:1,node3:1,node4:1,
clustering coefficient:0.000000
the shortest path between 1 and 3: 4
Path: 1 0 3
diameter:7
radius:4

case 2:
connected: 1
degree distribution:
node0:1,node1:3,node2:2,node3:3,node4:3,node5:1,node6:3,
clustering coefficient:0.238095
the shortest path between 1 and 3: 7
Path: 1 2 3
diameter:16
radius:9

case 3:
connected: 0
degree distribution:
node0:3,node1:2,node2:2,node3:3,node4:4,node5:2,node6:4,node7:2,node8:1,node9:1,
clustering coefficient:0.600000

Process finished with exit code 0

```

```
Run: metro x
D:\Programs\data-struct\experiments\ex4\src\cmake-build-debug\metro.exe
这个图是联通的。
线路图中换乘线路最多的站点是车公庙，共有 8 条线路通过。
该线路图的直径是 110，半径是 58。
从大学城站到机场站最少需要 32 分钟，路径：
大学城 --> 西丽 --> 留仙洞 --> 兴东 --> 洪浪北 --> 灵芝 --> 翻身 --> 宝安中心 --> 新安 --> 前海湾 --> 宝安 --> 碧海湾 --> 机场
Process finished with exit code 0
```

5 总结

在本次实验中，手动构建了：

1. C++容器版链式前向星
2. 优先队列
3. 带路径记录的和链式前向星结合的Dijkstra算法

在构建这些小项目时收获了很多，比如如何提高代码复用性，如何巧妙适当地解耦代码逻辑，如何用特殊算法提升算法效率。

同时提高了自己的动手实践能力，为将来工作打下良好基础。