华东师范大学计算机科学与技术学院上机实践报告

课程名称：数据结构	**年级**：2022 级	**上机实践成绩**：
指导教师：金健	**姓名**：岳锦鹏	**上机实践时间**：2 学时
上机实践名称：第六章作业	**学号**：10213903403	**上机实践日期**：2023/11/24

一、 实验目的

1. 使用第六章的知识解决图的综合问题。

二、 实验内容

1. 某国有 n 个城市，他们之间没有路连通，因此交通十分不便，现决定修建公路，任务由各个城市共同完成。修建分为 n 轮，在每一轮，每个城市选择一个与它最近的城市，申请修往该城市的公路，政府负责审批这些申请决定是否同意修建。

   政府审批规则如下：

   I. 如果两个城市申请修建同一条公路，则让后申请修建的城市重新选择未申请修建的公路。

   II. 如果三个或以上城市申请修建的公路成环，即 A 申请修建 AB，B 申请修建 BC，C 申请修建 CA，则政府将否决其中最短的一条公路的修建申请。

   III. 其他情况一律同意。

   一轮修建后，若干城市将会直接或间接相连，连通的城市被看作一个 league，下一轮修建中，league 被看作为一个城市。当所有的城市都连通时，则修建完毕。根据上述规则计算需要修建的公路的总长度。

2. 每个城市的规模大小是不同的（在初始需要赋值），在每个城市申请修建通往其他城市的道路时，需要查看城市规模系数 α，若规模小于目标城市，则政府拒绝修建，若两个城市规模相等，则修建过后两座城市形成的 league 城市规模系数 α 增一，若大于目标城市，则同意修建，规模系数不变。当城市规模系数 α 到达限定系数 β 后，我们认为该 league 已经到达“稳定”，不再与任何其他城市修建公路。根据上述规则计算需要修建的公路的总长度。

三、 实验原理

1. 程序设计原理。

四、 实验步骤

1. 问题抽象
2. 编写程序
3. 调试程序
4. 完善总结

五、 调试过程、结果和分析

1. 子类继承父类的时候应该不需要手动调用父类的析构函数，因为编译器会自动调用；
2. 调试带有随机性质的程序要先把随机数种子固定下来，调试完再用时间戳作为随机种子；
3. 类的保护属性在类外能直接访问？

六、 总结

1. 采用何种设计架构要综合考虑项目周期、系统复杂度、掌握熟练度等一系列因素，比如虽然这次的题可能有在时间复杂度或空间复杂度上更优的算法来解决，但是考虑到时间有限，还是使用了比较容易实现的思路；
2. CLion 的自动补全确实更加顺手，而且还能自动生成构造函数、析构函数等；
3. 此次也没有进行非常复杂的对象设计，只实现了并查集的类、图的类、复杂图的类；
4. 由于需要多次合并城市为一个联盟，所以使用了并查集；
5. 做题之前可以先考虑是否可以简化问题，比如这次的题的第二条规则就是无效的；
6. 输出的方式不一定要像图形界面，可以用文字表示，比如图可以用邻接矩阵、关系矩阵表示。

七、 附件

1. graph.cpp

```cpp
//
// Created by 423A35C7 on 2023-12-02.
//

#ifndef GRAPH_H
#define GRAPH_H

#include <algorithm>
#include <iostream>
#include <memory>
#include <random>
#include <cstring>
#include <list>
#include <map>

std::default_random_engine engine(time(nullptr));

class MergeFindSet {
    using int_ptr = std::unique_ptr<int>;

public:
    const int length;

```

```
24      explicit MergeFindSet(const int n)
25          : length(n), elements(new int[n]) {
26          memset(this->elements.get(), -1, n * sizeof(int));
27      }
28
29      int find(const int index) {
30          return elements.get()[index] == -1
31                     ? index
32                     : elements.get()[index] = find(elements.get()[index]);
33      }
34
35      void merge(const int a, const int b) {
36          elements.get()[find(b)] = find(a);
37      }
38
39      friend std::ostream& operator<<(std::ostream&out, const
        ↪  MergeFindSet&merge_find_set);
40
41  private:
42      int_ptr elements;
43  };
44
45  std::ostream& operator<<(std::ostream&out, const
    ↪  MergeFindSet&merge_find_set) {
46      for (int i = 0; i < merge_find_set.length; ++i) {
47          out << merge_find_set.elements.get()[i] << "\t";
48      }
49      return out;
50  }
51
52
53  /**
54   * \brief 打印高维类型
55   * \tparam T 最内层的类型
56   * \param high_dimension 高维指针，应为 int**, bool*** 等类似的类型
57   * \param length 长度（每一维的长度都应一样）
58   * \param dimensino_num 维度
59   */
60  template<typename T>
61  void print(void* high_dimension, const int length, const int dimensino_num
    ↪  = 1) {
62      void** temp = static_cast<void **>(high_dimension);
```

```
63      if (dimensino_num <= 1) {
64          for (int i = 0; i < length; ++i) {
65              std::cout << static_cast<T *>(high_dimension)[i] << "\t";
66          }
67          std::cout << std::endl;
68          return;
69      }
70      for (int i = 0; i < length; ++i) {
71          print<T>(temp[i], length, dimensino_num - 1);
72      }
73      std::cout << std::endl;
74  }
75
76  class Graph {
77  public:
78      ~Graph() {
79          for (int i = 0; i < this->node_num; ++i) {
80              delete this->adjacency[i];
81          }
82          delete this->adjacency;
83          for (int i = 0; i < this->node_num; ++i) {
84              delete this->incidence[i];
85          }
86          delete this->incidence;
87      }
88
89      explicit Graph(const int n)
90          : merge_find_set_(n) {
91          this->node_num = n;
92          std::uniform_int_distribution<int> get_random(1, 100);
93          this->adjacency = new int *[n];
94          this->incidence = new bool *[n];
95          for (int i = 0; i < n; ++i) {
96              this->adjacency[i] = new int[n];
97              this->incidence[i] = new bool[n];
98              memset(this->incidence[i], 0, n * sizeof(bool));
99              this->adjacency[i][i] = INT_MAX;
100             for (int j = 0; j < i; ++j) {
101                 this->adjacency[i][j] = this->adjacency[j][i] =
                    ↪ get_random(engine);
102             }
103         }
```

```
104         }
105
106         void print_adjacency() const {
107             print<int>(this->adjacency, this->node_num, 2);
108         }
109
110         void print_incidence() const {
111             print<bool>(this->incidence, this->node_num, 2);
112         }
113
114         void print_merge_find_set() {
115             std::cout << this->merge_find_set_ << std::endl;
116         }
117
118         bool same_league(const int start, const int target) {
119             return this->merge_find_set_.find(start) ==
                ↪  this->merge_find_set_.find(target);
120         }
121
122         /**
123          * \brief 把目标城市所在联盟合并到起始城市所在联盟中
124          * \param start 起始城市
125          * \param target 目标城市
126          */
127         void merge(const int start, const int target) {
128             this->merge_find_set_.merge(start, target);
129         }
130
131         /**
132          * \brief 开始记录关系矩阵是否被改变
133          */
134         void start_record_incidence() {
135             this->incidence_had_changed = false;
136         }
137
138         /**
139          * \brief 停止记录并返回关系矩阵是否被改变
140          * \return 关系矩阵是否被改变
141          */
142         bool stop_record_incidence() {
143             const bool temp = this->incidence_had_changed;
144             this->incidence_had_changed = false;
```

```
145         return temp;
146     }
147 
148     friend void one_turn(Graph*);
149 
150 protected:
151     /**
152      * \brief 邻接矩阵，表示任意两个城市之间的距离
153      */
154     int** adjacency; // 邻接矩阵
155     /**
156      * \brief 城市数量
157      */
158     int node_num;
159     /**
160      * \brief 关系矩阵，任意两个城市之间如果有直接的公路则为 true（是对称的，
         ↪ 反自反的关系）（不一定满足传递性）
161      */
162     bool** incidence; // 关系矩阵
163     bool incidence_had_changed;
164     MergeFindSet merge_find_set_;
165 };
166 
167 
168 class ComplexGraph : public Graph {
169 public:
170     ~ComplexGraph() {
171         // this->~Graph(); // 好像析构函数会自动调用，不需要手动调用
172         delete this->alpha;
173     }
174 
175     explicit ComplexGraph(const int n)
176         : Graph(n) {
177         this->alpha = new int[n];
178         std::uniform_int_distribution<int> get_random(1, n / 2);
179         for (int i = 0; i < n; ++i) {
180             this->alpha[i] = get_random(engine);
181         }
182     }
183 
184     void print_alpha() {
185         // print<int>(this->alpha, this->node_num, 1);
```

```
186         std::map<int, std::list<int>> index_to_league;
187         for (int i = 0; i < this->node_num; ++i) {
188             int ancestor = this->merge_find_set_.find(i);
189             if (!index_to_league.contains(ancestor)) {
190                 index_to_league.insert({ancestor, {}});
191             }
192             index_to_league[ancestor].push_back(i);
193         }
194         for (const auto&[ancestor, child]: index_to_league) {
195             std::cout << " 城市联盟 { ";
196             for (auto value: child) {
197                 std::cout << value << " ";
198             }
199             std::cout << " } 的规模系数为 ";
200             std::cout << this->get_alpha(ancestor) << std::endl;
201         }
202     }
203
204     void set_beta(const int beta) {
205         this->beta = beta;
206     }
207
208     /**
209      * \brief 获取城市所在联盟的规模系数
210      * \param index 城市序号
211      * \return 城市所在联盟的规模系数
212      */
213     [[nodiscard]] int get_alpha(const int index) {
214         return this->alpha[this->merge_find_set_.find(index)];
215     }
216
217     /**
218      * \brief 城市所在联盟的规模系数增加 1
219      * \param index 城市序号
220      */
221     void increase_alpha(const int index) {
222         this->alpha[this->merge_find_set_.find(index)]++;
223     }
224
225     friend void one_turn(ComplexGraph*);
226
227 private:
```

```
228     /**
229      * \brief 每个城市的规模系数
230      */
231     int* alpha;
232     int beta = 0; // 限定系数
233 };
234 
235 #endif //GRAPH_H
```

2. 第六章作业 1.cpp

```
1  /*
2   * 初步了解后发现实现获取最短的距离有三种方法：
3   * 1. 使用标准库的 merge 或者 inplace_merge，也就是归并排序；
4   * 2. 使用标准库的 priority_queue，也就是优先级队列，也就是堆；
5   * 3. 使用标准库的 multiset，也就是允许重复元素的集合（好像是用红黑树实现的）
6   * 由于这里每次只需要获取最短的路径，因此优先级队列可能是比较好的选择。
7   * （但是空间复杂度可能会比较高，因为 $n$ 个节点最多需要存 $n^2$ 条边，而实际上
    ↪  
8   * 最小生成树只要 n-1 条边就可以全部连通，优先级队列无法把长度过大的不可能选中
    ↪  的边排除）
9   * 但是优先级队列每次好像只能一个一个加，那还是用 merge 吧。
10  */
11 
12 /*
13  * 可以证明第二条规则无效，证明如下：
14  * 若 A、B、C 三个城市存在环，即 A 申请修建 AB，B 申请修建 BC，C 申请修建 CA，那
    ↪  么根据每个城市
15  * 只会选择与它最近的城市修建公路，则必定有 AB < AC，BC < BA，CA < CB，因此
16  * AB < AC < BC < AB，而 AB < AB 是不可能的，因此不会存在这样的情况；
17  * 同理，可以证明 n 个城市（n > 2）必定不存在环。
18  */
19 
20 /*
21  * 由于第二条规则无效，在第一题中，政府可以看做永远同意修建，因此，n 个城市一轮后
    ↪  就会修建了
22  * n 条公路，而 n-1 条公路就能使城市全部连通，因此一轮结束后城市就已全部连通。
23  */
24 
25 #include <iostream>
26 #include "graph.hpp"
```

```cpp

// 一轮
void one_turn(Graph* graph) {
    for (int start = 0; start < graph->node_num; ++start) {
        int target;
        while (true) {
            int* target_ptr = std::min_element(graph->adjacency[start],
                                               graph->adjacency[start] +
                                               graph->node_num);
            // 如果最小的目标城市距离都是 INT_MAX，则说明 start 与所有城市都已连通，则不再修建
            if (*target_ptr == INT_MAX) {
                return;
            }
            target = target_ptr - graph->adjacency[start];
            // 如果在同一个城市联盟内，那么把这个目标城市排除，在剩下的目标城市中继续
            if (graph->same_league(start, target)) {
                graph->adjacency[start][target] = INT_MAX;
                continue;
            }
            // 如果已经修建过了，则换个目标城市
            if (graph->incidence[start][target]) {
                continue;
            }
            // 不需要考虑三个或以上成环的情况
            break;
        }
        // 修建公路，即在关系矩阵上将相应的行列置为 true
        graph->incidence[start][target] = graph->incidence[target][start]
        = true;
        // 把目标城市所在联盟合并到起始城市所在联盟中
        graph->merge(start, target);
    }
}

int main() {
    int n;
    std::cout << " 输入城市的个数: ";
    std::cin >> n;
    auto graph = Graph(n);
    std::cout << " 初始的距离的邻接矩阵为: " << std::endl;
```

```
65      graph.print_adjacency();
66
67      one_turn(&graph);
68      std::cout << " 第一轮后的关系矩阵如下: " << std::endl;
69      graph.print_incidence();
70      std::cout << " 并查集如下: " << std::endl;
71      graph.print_merge_find_set();
72      std::cout << " 可以看到, 一轮后就已经全部连通。" << std::endl;
73      return 0;
74  }
75
76  // 输入城市的个数: 5
77  // 初始的距离的邻接矩阵为:
78  // 2147483647      69      19      14      90
79  // 69      2147483647      66      82      66
80  // 19      66      2147483647      48      8
81  // 14      82      48      2147483647      22
82  // 90      66      8       22      2147483647
83  //
84  // 第一轮后的关系矩阵如下:
85  // 0       0       0       1       0
86  // 0       0       1       0       0
87  // 0       1       0       0       1
88  // 1       0       0       0       1
89  // 0       0       1       1       0
90  //
91  // 并查集如下:
92  // -1      0       0       0       0
93  // 可以看到, 一轮后就已经全部连通。
```

3. 第六章作业 2.cpp

```
1   //
2   // Created by 423A35C7 on 2023-12-02.
3   //
4
5   #include <iostream>
6   #include "graph.hpp"
7   // 这次执行一轮后不一定结束了,
8   void one_turn(ComplexGraph* graph) {
9       for (int start = 0; start < graph->node_num; ++start) {
```

```
        int target;
        while (true) {
            int* target_ptr = std::min_element(graph->adjacency[start],
                                               graph->adjacency[start] +
                                               ↪ graph->node_num);
            // 如果最小的目标城市距离都是 INT_MAX，则说明 start 与所有城市都
            ↪ 已连通，则不再修建
            if (*target_ptr == INT_MAX) {
                return;
            }
            target = target_ptr - graph->adjacency[start];
            // 如果在同一个城市联盟内，那么把这个目标城市排除，在剩下的目标城
            ↪ 市中继续
            if (graph->same_league(start, target)) {
                graph->adjacency[start][target] = INT_MAX;
                continue;
            }
            // 如果已经修建过了，则换个目标城市
            if (graph->incidence[start][target]) {
                continue;
            }
            // 不需要考虑三个或以上成环的情况
            break;
        }
        const int start_alpha = graph->get_alpha(start);
        const int target_alpha = graph->get_alpha(target);
        // 当城市规模系数 到达限定系数 后，我们认为该 league
        // 已经到达“稳定”，不再与任何其他城市修建公路。
        if (start_alpha >= graph->beta || target_alpha >= graph->beta) {
            continue;
        }
        if (start_alpha < target_alpha) {
            // 规模小于目标城市，则政府拒绝修建
            continue;
        }
        else if (start_alpha == target_alpha) {
            // 若两个城市规模相等，则修建过后两座城市形成的 league 城市规模系
            ↪ 数 增一
            graph->increase_alpha(start);
        }
        else if (start_alpha > target_alpha) {
            // 若大于目标城市，则同意修建，规模系数不变
```

```
48              ;
49          }
50          // 修建公路，即在关系矩阵上将相应的行列置为 true
51          graph->incidence[start][target] = graph->incidence[target][start]
            ↪  = true;
52          graph->incidence_had_changed = true; // 保护字段为什么能直接访问？
53          // 把目标城市所在联盟合并到起始城市所在联盟中
54          graph->merge(start, target);
55      }
56  }
57
58  int main() {
59      int n, beta;
60
61      std::cout << " 输入城市的个数： ";
62      std::cin >> n;
63      auto graph = ComplexGraph(n);
64      std::cout << " 初始的城市规模为： " << std::endl;
65      graph.print_alpha();
66      std::cout << " 请输入限定系数 beta: ";
67      std::cin >> beta;
68      graph.set_beta(beta);
69
70      std::cout << " 初始的距离的邻接矩阵为： " << std::endl;
71      graph.print_adjacency();
72
73      for (int turn_num = 1; ; turn_num++) {
74          graph.start_record_incidence();
75          one_turn(&graph);
76          // 当关系矩阵不再被改变，也就是说明不再修桥了，则说明达到稳定
77          if (!graph.stop_record_incidence()) {
78              break;
79          }
80          std::cout << " 第" << turn_num << " 轮后的关系矩阵如下： " <<
            ↪  std::endl;
81          graph.print_incidence();
82          std::cout << " 规模系数如下： " << std::endl;
83          graph.print_alpha();
84          std::cout << " 并查集如下： " << std::endl;
85          graph.print_merge_find_set();
86          std::cout << std::endl;
87      }
```

```
88      std::cout << " 已经达到稳定" << std::endl;
89      return 0;
90  }
91
92  // 输入城市的个数: 10
93  // 初始的城市规模为:
94  // 城市联盟 { 0 } 的规模系数为 2
95  // 城市联盟 { 1 } 的规模系数为 5
96  // 城市联盟 { 2 } 的规模系数为 2
97  // 城市联盟 { 3 } 的规模系数为 1
98  // 城市联盟 { 4 } 的规模系数为 3
99  // 城市联盟 { 5 } 的规模系数为 4
100 // 城市联盟 { 6 } 的规模系数为 2
101 // 城市联盟 { 7 } 的规模系数为 2
102 // 城市联盟 { 8 } 的规模系数为 4
103 // 城市联盟 { 9 } 的规模系数为 3
104 // 请输入限定系数 beta: 5
105 // 初始的距离的邻接矩阵为:
106 // 2147483647    63    39    34    97    93    35    42
    ↪  68    29
107 // 63    2147483647    11    100    82    48    31    2
    ↪  33    58
108 // 39    11    2147483647    85    100    27    47    97
    ↪  86    91
109 // 34    100    85    2147483647    15    32    16    77
    ↪  84    29
110 // 97    82    100    15    2147483647    4    53    63
    ↪  54    29
111 // 93    48    27    32    4    2147483647    25    85
    ↪  86    58
112 // 35    31    47    16    53    25    2147483647    97
    ↪  27    88
113 // 42    2    97    77    63    85    97    2147483647
    ↪  61    91
114 // 68    33    86    84    54    86    27    61
    ↪  2147483647    66
115 // 29    58    91    29    29    58    88    91    66
    ↪  2147483647
116 //
117 // 第 1 轮后的关系矩阵如下:
118 // 0    0    0    0    0    0    0    0    0
    ↪  1
```

```
119 // 0       0       0       0       0       0       0       0       0
    ↪  0
120 // 0       0       0       0       0       0       0       0       0
    ↪  0
121 // 0       0       0       0       0       0       1       0       0
    ↪  0
122 // 0       0       0       0       0       1       0       0       0
    ↪  0
123 // 0       0       0       0       1       0       0       0       0
    ↪  0
124 // 0       0       0       1       0       0       0       0       1
    ↪  0
125 // 0       0       0       0       0       0       0       0       0
    ↪  0
126 // 0       0       0       0       0       0       1       0       0
    ↪  0
127 // 1       0       0       0       0       0       0       0       0
    ↪  0
128 //
129 // 规模系数如下:
130 // 城市联盟 { 1 } 的规模系数为 5
131 // 城市联盟 { 2 } 的规模系数为 2
132 // 城市联盟 { 4 5 } 的规模系数为 4
133 // 城市联盟 { 7 } 的规模系数为 2
134 // 城市联盟 { 3 6 8 } 的规模系数为 4
135 // 城市联盟 { 0 9 } 的规模系数为 3
136 // 并查集如下:
137 // 9       -1      -1      8       5       -1      8       -1      -1
    ↪  -1
138 //
139 // 第 2 轮后的关系矩阵如下:
140 // 0       0       0       0       0       0       0       0       0
    ↪  1
141 // 0       0       0       0       0       0       0       0       0
    ↪  0
142 // 0       0       0       0       0       0       0       0       0
    ↪  0
143 // 0       0       0       0       1       0       1       0       0
    ↪  0
144 // 0       0       0       1       0       1       0       0       0
    ↪  0
```

```
// 0       0       0       0       1       0       0       0       0
↪  0
// 0       0       0       1       0       0       0       0       1
↪  0
// 0       0       0       0       0       0       0       0       0
↪  0
// 0       0       0       0       0       0       1       0       0
↪  0
// 1       0       0       0       0       0       0       0       0
↪  0
//
// 规模系数如下:
// 城市联盟 { 1 } 的规模系数为 5
// 城市联盟 { 2 } 的规模系数为 2
// 城市联盟 { 7 } 的规模系数为 2
// 城市联盟 { 3 4 5 6 8 } 的规模系数为 5
// 城市联盟 { 0 9 } 的规模系数为 3
// 并查集如下:
// 9       -1      -1      8       8       8       8       -1      -1
↪  -1
//
// 已经达到稳定
```

4. CMakeLists.txt

```
cmake_minimum_required(VERSION 3.26)
project(chapter6)

set(CMAKE_CXX_STANDARD 23)

add_executable(graph1 第六章作业 1.cpp)
add_executable(graph2 第六章作业 2.cpp)
SET(EXECUTABLE_OUTPUT_PATH R:/)

set(CMAKE_CXX_FLAGS_RELEASE -fexec-charset=GBK)
```