山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202300130183	姓名： 宋浩宇	班级：23 级人工智能班
实验题目：2024 数据结构--数据/智能 实验 13 最小生成树		
实验学时：2	实验日期：2024/12/11	

实验目的：

1. 掌握堆结构的定义与实现；
2. 掌握堆结构的使用。

软件开发工具：

1. visual studio code 2022（使用 C/C++、C/C++ Extension Pack、C/C++ Themes 插件）
2. mingw64 工具包

1. 实验内容

完成 2024 数据结构--数据/智能 实验 13 最小生成树 A：Prim 算法和 2024 数据结构--数据/智能 实验 13 最小生成树 B ： Kruskal 算法。

1. 建图并实现 prim 算法
2. 建图并实现 Kruskal 算法

2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

本题使用的数据结构为无向有权图，更特殊的说，是无向有权连通图（参照最小生成树的定义，最小生成树需要在连通图中生成）我们图的构建可以完全按照实验 12 的方式来构建，只需要在边上加一个 weight 值即可。

我们分别来描述 Prim 算法和 Kruskal 算法

这两个算法的核心思路都是相对简单的，但是因为这个题的数据量较大，朴素的 Prim 算法和 Kruskal 算法都会超时，因此需要相应的优化，我们先描述算法，在描述优化。

对于 Prim 算法，我们将所有的点分为两个集合，一个是待选点，一个是已选点，我们随便从待选点集合中挑选一个点放入已选点集合，然后遍历已选点的每一条边，选择其中权值最小且连接的另一个点在待选集合中的边，将这个边的权值加入最终结果，将这个边连接的另一个点从待选点结合中放入已选点集合中，再重复以上过程直到待选点集合为空。这是核心思路，在实际实现的时候为了提高速度使用的是一个 bool 数组来存储已选和待选的情况。

对于优化方式，Prim 算法的优化主要是使用了优先级队列来处理边，当我们把新的点加入这个集合的时候，我们把这个点的边直接插入优先级队列，然后在每次循环中都取出这个优先级队列的队列头的边，然后如果这个边的两个点都已在已选点集合里，则 pop 并进入下一轮循环，如果不在则将这个边链接的那个点加入已选点集合，这个加入的点的边都加入这个优先级队列。用这种方式就能将算法的时间复杂度从 O(n^2)降低至 O(nlogn)。

对于 Kruskal 算法，我们将所有的边都放入一个线性表里，在对这个线性表进行排序获得一个根据权值升序排列的边的线性表（考虑到代码编写难易度以及堆排序和快速排序的平均时间复杂度相差并不大（O(nlog^2n)和 O(nlogn)）我们选择优先级队列来进行排序）。然后我们再依次取出这个表中权值最小的边，并将这个边对应的两个点编入一个组，且将这两个点都标为已选择。实际上对于每条边，有以下处理：如果这个边的两个点都已经被选择了，且在同一个组里，则 pop 掉这个边然后进入下一次循环，其余的情况，就是把这两个节点所在的组合并，即把这两个节点的以及与他们相连的已选择的点的组号都改为同一个。只要重复上述过程直到把边全都消耗掉，或者所有点都被加入到同一个组里，这个树就构建好了。

关于 Kruskal 算法的优化，主要需要并查集这个数据结构，实际上我们在实际编写的时候只是使用了并查集的思想。使用并查集的方式就可以大大提高这种不重复集合合并的速度。具

体实现方式为，我们使用一个数组存储每一个节点的父节点，且在初始化的时候都设置为其本身。然后当判断一个边的两个点时，我们让这两个点的父节点都去寻找自己的父节点直到找到父节点的父节点是自己的时候停止，这个就是这两个点对应的根节点，然后我们把这两个节点的父节点都改成对应的根节点，然后我们就可以与上文一样对这个根节点进行判断，这个根节点就对应了这两个节点所在的组，如果这两个点的根节点一致，说明它们已经连通了，则 pop 掉这个边进入下一次循环，如果不一致，则可以进行链接，连接方式为将这两个节点的根节点的父节点都设置为这两个父节点中索引较小的那一个（实际索引较大的情况也是等价的），只要循环以上过程，直到把边全都消耗掉，或者所有点都被加入到同一个组里，这个树就构建好了。同时因为我们使用了并查集的思想，且进行了路径收缩的方法，实际在判断这两个节点是否为同一组，以及合并两个组的时候效率是极高的因此我们的 Kruskal 的时间复杂度也是成功降低了。

3. 测试结果（测试输入，测试输出）

对于 A 题，测试输入：

```
7 12
1 2 9
1 5 2
1 6 3
2 3 5
2 6 7
3 4 6
3 7 3
4 5 6
4 7 2
5 6 3
5 7 6
6 7 1
```

输出为：

```
16
```

对于 B 题，测试输入：

```
7 12
1 2 9
1 5 2
1 6 3
2 3 5
2 6 7
3 4 6
3 7 3
4 5 6
4 7 2
5 6 3
5 7 6
6 7 1
```

输出为：

```
16
```

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

从测试结果来看，我们的算法成功解决了这个问题。但是，本题我们存在的主要问题是，我

们的算法默认了题目给的数据是合法的，即题目给的图是一个强连通图，因此我们并没有检测这个图是否是强连通图，是否存在最小生成树。解决方案就是在进行最小生成树的计算之前，先检测这个图是否是强连通图，能够确定这个图存在最小生成树之后在进行最小生成树的计算。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

A 题 Prim 算法

```
/*2024 数据结构--数据智能 实验 13 最小生成树 A Prime 算法*/

#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>


using namespace std;

struct GraphEdgeWithCost
{
    size_t from;
    size_t to;
    size_t cost;
    bool operator<(const GraphEdgeWithCost& other) const;
    bool operator==(const GraphEdgeWithCost& other) const;
    bool operator!=(const GraphEdgeWithCost& other) const;
    bool operator>(const GraphEdgeWithCost& other) const;
};

bool GraphEdgeWithCost::operator<(const GraphEdgeWithCost& other) c
onst
{
    return cost < other.cost;
}

bool GraphEdgeWithCost::operator==(const GraphEdgeWithCost& other)
const
{
    return from == other.from && to == other.to && cost == other.co
st;
}

bool GraphEdgeWithCost::operator!=(const GraphEdgeWithCost& other)
const
{
    return !(*this == other);
}

```

```
37.  bool GraphEdgeWithCost::operator>(const GraphEdgeWithCost& other) c
   onst
38.  {
39.      return cost > other.cost;
40.  }
41.
42.  template<typename T>
43.  struct GraphNode
44.  {
45.      T data;
46.      size_t id;
47.      vector<GraphEdgeWithCost> edges;
48.      vector<size_t> neighbors;
49.      size_t group;
50.      void addEdge(size_t to, size_t cost);
51.      void deleteEdge(size_t to);
52.      void printId();
53.      void printEdges();
54.      void printData();
55.      void print();
56.  };
57.
58.  template<typename T>
59.  void GraphNode<T>::print()
60.  {
61.      printId();
62.      printData();
63.      printEdges();
64.  }
65.
66.  template<typename T>
67.  void GraphNode<T>::printId()
68.  {
69.      cout << "NODE ID: " << id << endl;
70.  }
71.
72.  template<typename T>
73.  void GraphNode<T>::printData()
74.  {
75.      cout <<"NODE DATA: " << data << endl;
76.  }
77.
78.  template<typename T>
79.  void GraphNode<T>::printEdges()
80.  {
81.      cout << "NODE EDGES: " << endl;
```

```
82.       for (auto& e : edges)
83.       {
84.           cout << e.from << "->" << e.to << " (" << e.cost << ")" <<
   endl;
85.       }
86.   }
87.
88.
89.   template<typename T>
90.   void GraphNode<T>::addEdge(size_t to, size_t cost)
91.   {
92.       edges.push_back({ id, to, cost });
93.       neighbors.push_back(to);
94.   }
95.
96.   template<typename T>
97.   void GraphNode<T>::deleteEdge(size_t to)
98.   {
99.       auto it = find_if(edges.begin(), edges.end(), [to](const GraphE
   dgeWithCost& e) { return e.to == to; });
100.      if (it!= edges.end())
101.      {
102.          edges.erase(it);
103.          auto it2 = find(neighbors.begin(), neighbors.end(), to);
104.          if (it2!= neighbors.end())
105.          {
106.              neighbors.erase(it2);
107.          }
108.      }
109.  }
110.
111.
112.  template<typename T>
113.  class GraphWithCostEdges        //实际是无向图
114.  {
115.  private:
116.      vector<GraphNode<T>> nodes;
117.      bool connected;
118.  public:
119.      GraphWithCostEdges() {}
120.      GraphWithCostEdges(size_t n);
121.      void addEdge(size_t from, size_t to, size_t cost);
122.      void deleteEdge(size_t from, size_t to);
123.      void printGraph();
124.
125.      bool isConnected();
```

```
126.     void checkConnected();
127. 
128.     size_t prim();
129.     size_t kruskal();
130. };
131. 
132. template<typename T>
133. GraphWithCostEdges<T>::GraphWithCostEdges(size_t n)
134. {
135.     connected = true;
136.     nodes.resize(n + 1);
137.     for (size_t i = 1; i <= n; i++)
138.     {
139.         nodes[i].id = i;
140.         nodes[i].data = i;
141.     }
142. }
143. 
144. template<typename T>
145. void GraphWithCostEdges<T>::addEdge(size_t from, size_t to, size_t
     cost)
146. {
147.     nodes[from].addEdge(to, cost);
148.     nodes[to].addEdge(from, cost);
149. }
150. 
151. template<typename T>
152. void GraphWithCostEdges<T>::deleteEdge(size_t from, size_t to)
153. {
154.     nodes[from].deleteEdge(to);
155.     nodes[to].deleteEdge(from);
156. }
157. 
158. template<typename T>
159. void GraphWithCostEdges<T>::printGraph()
160. {
161.     for (auto& node : nodes)
162.     {
163.         node.print();
164.     }
165. }
166. 
167. template<typename T>
168. bool GraphWithCostEdges<T>::isConnected()
169. {
170.     return connected;
```

```
171. }
172.
173. template<typename T>
174. size_t GraphWithCostEdges<T>::prim()
175. {
176.     if (!isConnected())
177.     {
178.         return 0;
179.     }
180.     size_t mst_cost = 0;
181.     //下为朴素 Prim 算法
182.     /*
183.     vector<bool> visited(nodes.size(), false);
184.     vector<size_t> selected;
185.     selected.push_back(1);
186.     visited[0] = true;
187.     visited[1] = true;
188.     while (selected.size() < nodes.size() - 1)
189.     {
190.         size_t min_cost = numeric_limits<size_t>::max();
191.         GraphEdgeWithCost* min_edge = nullptr;
192.         for (auto& node_index : selected)
193.         {
194.             for (auto& edge : nodes[node_index].edges)
195.             {
196.                 if (!visited[edge.to])
197.                 {
198.                     if (edge.cost < min_cost)
199.                     {
200.                         min_cost = edge.cost;
201.                         min_edge = &edge;
202.                     }
203.                 }
204.             }
205.         }
206.         selected.push_back(min_edge->to);
207.         visited[min_edge->to] = true;
208.         mst_cost += min_edge->cost;
209.         // cout << "Selected: " << min_edge->from << "->" << min_ed
    ge->to << " (" << min_edge->cost << ")" << endl;
210.     }
211.     */
212.     //下为堆优化 Prim 算法
213.     priority_queue<GraphEdgeWithCost, vector<GraphEdgeWithCost>, gr
    eater<GraphEdgeWithCost>> edges;
214.     vector<bool> visited(nodes.size(), false);
```

```
215.     visited[0] = true;
216.     visited[1] = true;
217.     vector<size_t> selected;
218.     selected.push_back(1);
219.     for (auto& edge : nodes[1].edges)
220.     {
221.         edges.push(edge);
222.     }
223.     while (selected.size() < nodes.size() - 1)
224.     {
225.         if (edges.empty())
226.         {
227.             break;
228.         }
229.         const GraphEdgeWithCost& selected_edge = edges.top();
230.         if (visited[selected_edge.to])
231.         {
232.             edges.pop();
233.             continue;
234.         }
235.         else
236.         {
237.             selected.push_back(selected_edge.to);
238.             visited[selected_edge.to] = true;
239.             mst_cost += selected_edge.cost;
240.             for (auto& edge : nodes[selected_edge.to].edges)
241.             {
242.                 if (!visited[edge.to])
243.                 {
244.                     edges.push(edge);
245.                 }
246.             }
247.         }
248.     }
249.
250.     return mst_cost;
251. }
252.
253. template<typename T>
254. size_t GraphWithCostEdges<T>::kruskal()
255. {
256.     if (!isConnected())
257.     {
258.         return 0;
259.     }
260.     size_t mst_cost = 0;
```

```
261.
262.      priority_queue<GraphEdgeWithCost, vector<GraphEdgeWithCost>, gr
   eater<GraphEdgeWithCost>> edges;
263.      vector<bool> visited(nodes.size(), false);
264.      while (!edges.empty())
265.      {
266.          if (visited[edges.top().from] && visited[edges.top().to])
267.          {
268.              edges.pop();
269.              continue;
270.          }
271.          const GraphEdgeWithCost& edge = edges.top();
272.          visited[edge.from] = true;
273.          visited[edge.to] = true;
274.          mst_cost += edge.cost;
275.          edges.pop();
276.      }
277.      return mst_cost;
278.  }
279.
280.  template<typename T>
281.  void GraphWithCostEdges<T>::checkConnected()
282.  {
283.      queue<size_t> q;
284.      q.push(1);
285.      vector<bool> visited(nodes.size(), false);
286.      visited[0] = true;
287.      visited[1] = true;
288.      while (!q.empty())
289.      {
290.          size_t u = q.front();
291.          q.pop();
292.          for (auto& e : nodes[u].edges)
293.          {
294.              if (!visited[e.to])
295.              {
296.                  visited[e.to] = true;
297.                  q.push(e.to);
298.              }
299.          }
300.      }
301.
302.      for (bool v : visited)
303.      {
304.          if (!v)
305.          {
```

```
                connected = false;
                return;
            }
        }
        connected = true;
    }

class Solution
{
public:
    void solve();
};

void Solution::solve()
{
    ios::sync_with_stdio(false);
    size_t n, e;
    cin >> n >> e;
    GraphWithCostEdges<int> graph(n);
    for (size_t i = 0; i < e; i++)
    {
        size_t from, to, cost;
        cin >> from >> to >> cost;
        graph.addEdge(from, to, cost);
    }
    // graph.printGraph();
    // graph.checkConnected();
    // cout << "Is connected: " << graph.isConnected() << endl;
    // cout << "MST cost: " << graph.prim() << endl;
    cout << graph.prim() << endl;
    // cout << graph.kruskal() << endl;
}

int main()
{
    Solution solution;
    solution.solve();
    return 0;
}
```

B 题 Kruskal 算法

```
/*2024 数据结构--数据智能 实验 13 最小生成树 B Kruskal 算法*/


#include <iostream>
#include <vector>
```

```
6.  #include <queue>
7.  #include <algorithm>
8.
9.  using namespace std;
10.
11.  struct GraphEdgeWithCost
12.  {
13.      size_t from;
14.      size_t to;
15.      size_t cost;
16.      bool operator<(const GraphEdgeWithCost& other) const;
17.      bool operator==(const GraphEdgeWithCost& other) const;
18.      bool operator!=(const GraphEdgeWithCost& other) const;
19.      bool operator>(const GraphEdgeWithCost& other) const;
20.  };
21.
22.  bool GraphEdgeWithCost::operator<(const GraphEdgeWithCost& other) c
   onst
23.  {
24.      return cost < other.cost;
25.  }
26.
27.  bool GraphEdgeWithCost::operator==(const GraphEdgeWithCost& other)
   const
28.  {
29.      return from == other.from && to == other.to && cost == other.co
   st;
30.  }
31.
32.  bool GraphEdgeWithCost::operator!=(const GraphEdgeWithCost& other)
   const
33.  {
34.      return !(*this == other);
35.  }
36.
37.  bool GraphEdgeWithCost::operator>(const GraphEdgeWithCost& other) c
   onst
38.  {
39.      return cost > other.cost;
40.  }
41.
42.  template<typename T>
43.  struct GraphNode
44.  {
45.
46.      GraphNode* prev; //用于链表实现并查集
```

```
47.         size_t root;      //用于数组实现并查集
48.         T data;
49.         size_t id;
50.         vector<GraphEdgeWithCost> edges;
51.         vector<size_t> neighbors;
52.         size_t group;
53.         void addEdge(size_t to, size_t cost);
54.         void deleteEdge(size_t to);
55.         void printId();
56.         void printEdges();
57.         void printData();
58.         void print();
59.     };
60.     template<typename T>
61.     bool operator<(const GraphNode<T>& a, const GraphNode<T>& b)
62.     {
63.         return a.data < b.data;
64.     }
65.
66.
67.     template<typename T>
68.     void GraphNode<T>::print()
69.     {
70.         printId();
71.         printData();
72.         printEdges();
73.     }
74.
75.     template<typename T>
76.     void GraphNode<T>::printId()
77.     {
78.         cout << "NODE ID: " << id << endl;
79.     }
80.
81.     template<typename T>
82.     void GraphNode<T>::printData()
83.     {
84.         cout <<"NODE DATA: " << data << endl;
85.     }
86.
87.     template<typename T>
88.     void GraphNode<T>::printEdges()
89.     {
90.         cout << "NODE EDGES: " << endl;
91.         for (auto& e : edges)
92.         {
```

```
93.             cout << e.from << "->" << e.to << " (" << e.cost << ")" <<
    endl;
94.         }
95.     }
96.
97.
98.     template<typename T>
99.     void GraphNode<T>::addEdge(size_t to, size_t cost)
100.    {
101.        edges.push_back({ id, to, cost });
102.        neighbors.push_back(to);
103.    }
104.
105.    template<typename T>
106.    void GraphNode<T>::deleteEdge(size_t to)
107.    {
108.        auto it = find_if(edges.begin(), edges.end(), [to](const GraphE
    dgeWithCost& e) { return e.to == to; });
109.        if (it!= edges.end())
110.        {
111.            edges.erase(it);
112.            auto it2 = find(neighbors.begin(), neighbors.end(), to);
113.            if (it2!= neighbors.end())
114.            {
115.                neighbors.erase(it2);
116.            }
117.        }
118.    }
119.
120.
121.    template<typename T>
122.    class GraphWithCostEdges        //实际是无向图
123.    {
124.    private:
125.        vector<GraphNode<T>> nodes;
126.        bool connected;
127.        vector<GraphEdgeWithCost> graph_edges;
128.    public:
129.        GraphWithCostEdges() {}
130.        GraphWithCostEdges(size_t n);
131.        void addEdge(size_t from, size_t to, size_t cost);
132.        void deleteEdge(size_t from, size_t to);
133.        void printGraph();
134.
135.        bool isConnected();
136.        void checkConnected();
```

```
137.
138.     size_t prim();
139.     size_t kruskal();
140. };
141.
142. template<typename T>
143. GraphWithCostEdges<T>::GraphWithCostEdges(size_t n)
144. {
145.     connected = true;
146.     nodes.resize(n + 1);
147.     for (size_t i = 1; i <= n; i++)
148.     {
149.         nodes[i].id = i;
150.         nodes[i].data = i;
151.         nodes[i].prev = nullptr;
152.     }
153. }
154.
155. template<typename T>
156. void GraphWithCostEdges<T>::addEdge(size_t from, size_t to, size_t
    cost)
157. {
158.     nodes[from].addEdge(to, cost);
159.     nodes[to].addEdge(from, cost);
160.     graph_edges.push_back({ from, to, cost });
161. }
162.
163. template<typename T>
164. void GraphWithCostEdges<T>::deleteEdge(size_t from, size_t to)
165. {
166.     nodes[from].deleteEdge(to);
167.     nodes[to].deleteEdge(from);
168. }
169.
170. template<typename T>
171. void GraphWithCostEdges<T>::printGraph()
172. {
173.     for (auto& node : nodes)
174.     {
175.         node.print();
176.     }
177. }
178.
179. template<typename T>
180. bool GraphWithCostEdges<T>::isConnected()
181. {
```

```
182.     return connected;
183. }
184. 
185. template<typename T>
186. size_t GraphWithCostEdges<T>::prim()
187. {
188.     if (!isConnected())
189.     {
190.         return 0;
191.     }
192.     size_t mst_cost = 0;
193.     //下为朴素 Prim 算法
194.     /*
195.     vector<bool> visited(nodes.size(), false);
196.     vector<size_t> selected;
197.     selected.push_back(1);
198.     visited[0] = true;
199.     visited[1] = true;
200.     while (selected.size() < nodes.size() - 1)
201.     {
202.         size_t min_cost = numeric_limits<size_t>::max();
203.         GraphEdgeWithCost* min_edge = nullptr;
204.         for (auto& node_index : selected)
205.         {
206.             for (auto& edge : nodes[node_index].edges)
207.             {
208.                 if (!visited[edge.to])
209.                 {
210.                     if (edge.cost < min_cost)
211.                     {
212.                         min_cost = edge.cost;
213.                         min_edge = &edge;
214.                     }
215.                 }
216.             }
217.         }
218.         selected.push_back(min_edge->to);
219.         visited[min_edge->to] = true;
220.         mst_cost += min_edge->cost;
221.         // cout << "Selected: " << min_edge->from << "->" << min_ed
    ge->to << " (" << min_edge->cost << ")" << endl;
222.     }
223.     */
224.     //下为堆优化 Prim 算法
225.     priority_queue<GraphEdgeWithCost, vector<GraphEdgeWithCost>, gr
    eater<GraphEdgeWithCost>> edges;
```

```
226.     vector<bool> visited(nodes.size(), false);
227.     visited[0] = true;
228.     visited[1] = true;
229.     vector<size_t> selected;
230.     selected.push_back(1);
231.     for (auto& edge : nodes[1].edges)
232.     {
233.         edges.push(edge);
234.     }
235.     while (selected.size() < nodes.size() - 1)
236.     {
237.         if (edges.empty())
238.         {
239.             break;
240.         }
241.         const GraphEdgeWithCost& selected_edge = edges.top();
242.         if (visited[selected_edge.to])
243.         {
244.             edges.pop();
245.             continue;
246.         }
247.         else
248.         {
249.             selected.push_back(selected_edge.to);
250.             visited[selected_edge.to] = true;
251.             mst_cost += selected_edge.cost;
252.             for (auto& edge : nodes[selected_edge.to].edges)
253.             {
254.                 if (!visited[edge.to])
255.                 {
256.                     edges.push(edge);
257.                 }
258.             }
259.         }
260.     }
261.
262.     return mst_cost;
263. }
264.
265. template<typename T>
266. size_t GraphWithCostEdges<T>::kruskal()
267. {
268.     if (!isConnected())
269.     {
270.         return 0;
271.     }
```

```
272.     size_t mst_cost = 0;
273. 
274.     priority_queue<GraphEdgeWithCost, vector<GraphEdgeWithCost>, gr
   eater<GraphEdgeWithCost>> edges;
275.     vector<bool> visited(nodes.size(), false);
276. 
277.     for (auto edge : graph_edges)
278.     {
279.         edges.push(edge);
280.     }
281. 
282.     //分组式朴素 Kruskal 算法
283.     // for (auto& node : nodes)
284.     // {
285.     //     node.group = node.id;
286.     // }
287. 
288.     // while (!edges.empty())
289.     // {
290.     //     if ((visited[edges.top().from] && visited[edges.top().to
   ]) && (nodes[edges.top().from].group == nodes[edges.top().to].group))

291.     //     {
292.     //         edges.pop();
293.     //         continue;
294.     //     }
295.     //     const GraphEdgeWithCost& edge = edges.top();
296.     //     visited[edge.from] = true;
297.     //     visited[edge.to] = true;
298.     //     auto temp_from_group = nodes[edge.from].group;
299.     //     auto temp_to_group = nodes[edge.to].group;
300.     //     for (auto& node : nodes)
301.     //     {
302.     //         if (node.group == temp_from_group)
303.     //         {
304.     //             node.group = temp_to_group;
305.     //         }
306.     //     }
307.     //     // for (auto& node : nodes)
308.     //     // {
309.     //     //     cout << "node id: " << node.id << " group: " << n
   ode.group << endl;
310.     //     // }
311.     //     // cout << "Selected: " << edge.from << "->" << edge.to
   << " (" << edge.cost << ")" << endl;
312.     //     mst_cost += edge.cost;
```

```
313.      //      edges.pop();
314.      // }
315.
316.
317.      //并查集优化 Kruskal 算法
318.
319.      // for (size_t i = 1; i < nodes.size(); i++)
320.      // {
321.      //      nodes[i].prev = &nodes[i];
322.      // }
323.      // while (!edges.empty())
324.      // {
325.      //      const GraphEdgeWithCost& edge = edges.top();
326.      //      GraphNode<T>* from_node_prev = &nodes[edge.from];
327.      //      GraphNode<T>* to_node_prev = &nodes[edge.to];
328.      //      while (from_node_prev->prev != nullptr && from_node_prev
   ->prev != from_node_prev)
329.      //      {
330.      //          from_node_prev = from_node_prev->prev;
331.      //      }
332.      //      while (to_node_prev->prev != nullptr && to_node_prev->pr
   ev != to_node_prev)
333.      //      {
334.      //          to_node_prev = to_node_prev->prev;
335.      //      }
336.      //      nodes[edge.from].prev = to_node_prev;
337.      //      nodes[edge.to].prev = from_node_prev;
338.      //      if (from_node_prev == to_node_prev && from_node_prev !=
   nullptr && to_node_prev != nullptr)
339.      //      {
340.      //          edges.pop();
341.      //          continue;
342.      //      }
343.      //      // auto& from_node = nodes[edge.from];
344.      //      // auto& to_node = nodes[edge.to];
345.
346.      //      GraphNode<T>* root = new GraphNode<T>();
347.      //      root->prev = nullptr;
348.
349.      //      from_node_prev->prev = root;
350.      //      to_node_prev->prev = root;
351.
352.      //      mst_cost += edge.cost;
353.      //      edges.pop();
354.      //      // for (auto& node : nodes)
355.      //      // {
```

```
356.      //      //      cout << "node id: " << node.id << " group: " << n
   ode.group << endl;
357.      //      // }
358.      //      // cout << "Selected: " << edge.from << "->" << edge.to
   << " (" << edge.cost << ")" << endl;
359.      // }
360.
361.      //数组描述的并查集优化 Kruskal 算法
362.      int parent[nodes.size() + 1] = { -1 };
363.      for (size_t i = 1; i < nodes.size(); i++)
364.      {
365.          parent[i] = i; // -1 表示未访问，0 表示根节点
366.      }
367.      parent[0] = 0;
368.      parent[1] = 0;
369.      while (!edges.empty())
370.      {
371.          const GraphEdgeWithCost& edge = edges.top();
372.
373.          int from_root = edge.from;
374.          int to_root = edge.to;
375.
376.          while (parent[from_root] != from_root)
377.          {
378.              from_root = parent[from_root];
379.          }
380.          while (parent[to_root] != to_root)
381.          {
382.              to_root = parent[to_root];
383.          }
384.
385.          nodes[edge.from].root = from_root;
386.          nodes[edge.to].root = to_root;
387.
388.          if (from_root == to_root)
389.          {
390.              edges.pop();
391.              continue;
392.          }
393.
394.          int min_root = from_root < to_root ? from_root : to_root;
395.
396.          parent[from_root] = min_root;
397.          parent[to_root] = min_root;
398.
399.          mst_cost += edge.cost;
```

```
400.             edges.pop();
401.             // for (auto& node : nodes)
402.             // {
403.             //     cout << "node id: " << node.id << " group: " << node
    .group << endl;
404.             // }
405.             // cout << "Selected: " << edge.from << "->" << edge.to <<
    " (" << edge.cost << ")" << endl;
406.         }
407.
408.         return mst_cost;
409.     }
410.
411.     template<typename T>
412.     void GraphWithCostEdges<T>::checkConnected()
413.     {
414.         queue<size_t> q;
415.         q.push(1);
416.         vector<bool> visited(nodes.size(), false);
417.         visited[0] = true;
418.         visited[1] = true;
419.         while (!q.empty())
420.         {
421.             size_t u = q.front();
422.             q.pop();
423.             for (auto& e : nodes[u].edges)
424.             {
425.                 if (!visited[e.to])
426.                 {
427.                     visited[e.to] = true;
428.                     q.push(e.to);
429.                 }
430.             }
431.         }
432.
433.         for (bool v : visited)
434.         {
435.             if (!v)
436.             {
437.                 connected = false;
438.                 return;
439.             }
440.         }
441.         connected = true;
442.     }
443.
```

```
444.
445. class Solution
446. {
447. public:
448.     void solve();
449. };
450.
451. void Solution::solve()
452. {
453.     ios::sync_with_stdio(false);
454.     size_t n, e;
455.     cin >> n >> e;
456.     GraphWithCostEdges<int> graph(n);
457.     for (size_t i = 0; i < e; i++)
458.     {
459.         size_t from, to, cost;
460.         cin >> from >> to >> cost;
461.         graph.addEdge(from, to, cost);
462.     }
463.     // graph.printGraph();
464.     // graph.checkConnected();
465.     // cout << "Is connected: " << graph.isConnected() << endl;
466.     // cout << "MST cost: " << graph.prim() << endl;
467.     // cout << graph.prim() << endl;
468.     cout << graph.kruskal() << endl;
469. }
470.
471. int main()
472. {
473.     Solution solution;
474.     solution.solve();
475.     return 0;
476. }
```