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 12 图		
实验学时：2	实验日期：2024/12/4	

实验目的：

1.掌握图结构的定义与实现；

2.掌握图结构的使用。

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 12 图 A 图论基础

1.实现图类，包含功能：删除边，插入边

2.完成实验的七个操作（七行输出）

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是无向无权图。题目要求我们实现的七个操作分别是：

第一行输出图中有多少个连通分量

第二行输出所有连通子图中最小点的编号（升序），编号间用空格分隔

第三行输出从 s 点开始的 dfs 序列长度

第四行输出从 s 点开始的字典序最小的 dfs 序列

第五行输出从 t 点开始的 bfs 序列的长度

第六行输出从 t 点开始字典序最小的 bfs 序列

第七行输出从 s 点到 t 点的最短路径，若是不存在路径则输出-1

简单来说，题目需要我们实现一个 DFS、一个 BFS 以及一个字典序优先的 DFS、一个字典序优先的 BFS，一个获取连通分量的函数，一个获取连通子图中最小编号的函数，一个搜索最短路径的函数，因为在实现前两行要求的输出的时候实际使用到了 BFS 或者 DFS 的，因此我们先来描述 BFS 和 DFS 的实现方式。

首先需要声明的是，因为题目的第一行、第二行、第七行也需要用到 BFS 和 DFS 其中一种，考虑到这两种搜索算法的时间复杂度在进行无目标搜索时是一样的且在无向无权图中 Dijkstra 算法和 BFS 是一致的，我们选择用递归的方式实现 DFS，用循环的方式实现 BFS，这样可以简化代码，且保证 BFS 高效运行。

I. 关于 BFS：

因为我们是使用循环的方式来实现，故有如下内容。

在此处我们需要使用到队列数据结构。

我们创建一个队列，队列里存储的是我们将要处理的节点。

我们在创建一个 bool 数组，用于存储哪些节点是我们已经搜索到的。

我们在每一个循环里取出队列中的第一个节点，并对这个节点进行处理（即按照传入的函数指针进行处理），再对这个节点的字节点进行处理，把这个节点的子节点中还未搜索到的节点压入队列，并将这些节点标记为已经被搜索到。重复这个过程直到队列为空，则此时我们的无目标 BFS 结束。

而开始 BFS 需要进行的操作就是把起始节点压入队列并将其标记为已经搜索到了。

II. 关于 DFS：

因为我们是使用递归的方式来实现，故有如下内容。
我们在每一层 DFS 中都需要如下参数：该层 DFS 处理的节点索引、每一层 DFS 通用的用于标记节点是否被搜索到的 bool 数组、（可选）处理节点使用的函数指针
在每一层 DFS 中,我们先对该层进行处理的节点进行处理(即按照传入的函数指针进行处理)，再对这个节点的子节点进行处理，先把这个节点的子节点中未被搜索到的节点标记为已被搜索到，再让这些子节点进入下一层 DFS。如此下去，当递归结束的时候就是我们无目标 DFS 结束的时候。
III. 关于字典序优先。因为题目给我们的数据都是 int 类型的数字，因此我们只需要在 DFS 和 BFS 时对于当前处理的子节点序列进行排序再按照排序后的结果进行递归或压入队列的操作即可。
显然，我们只需要一个简单的计数函数，将它的指针分别传入 DFS 和 BFS 就可以得到第三行和第五行的结果。
第四行和第六行的输出我们使用字典序 DFS 和字典序 BFS 再传入一个输出节点数据的函数指针即可。
而第一行要求的输出，即连通分量的个数，我们只需要开一个 bool 数组存储节点是否已被遍历到，然后遍历所有节点，如果这个节点没有被遍历过，就将连通分量的计数加 1，再通过 BFS 将与这个节点所有连通的节点都设为已被遍历过即可，经过这个过程我们可以轻易的得到图的连通分量的个数。同样的思路，我们用同样的方式处理第二行的输出，只需要开一个数组存储子连通图的最小节点，并在上文中每次进行搜索过程时找出这个子连通图中的最小节点并将这个节点存储到上述数组里，把这个数组排序后再输出即可。
最后是最短路径的搜索，我们是用最简单的 Dijkstra 算法来进行搜索，因为 Dijkstra 的原理，实际的搜索过程在无向无权图中与 BFS 是一致的，因此我们使用 BFS 的方式来搜索最短路径的长度。总之这个题虽然要求较多，但实际上并不困难。
3. 测试结果（测试输入，测试输出）
测试输入：

```
10 20 4 5
0 6 4
0 10 3
0 4 8
0 4 10
1 4 10
0 2 1
0 5 8
0 5 2
0 10 7
0 9 6
0 9 1
0 7 1
0 8 10
0 7 5
0 8 3
0 6 7
1 6 4
1 8 3
0 7 8
0 9 2
```

输出为：
1
1
10
4 8 5 2 1 7 6 9 10 3
10
5 2 7 8 1 9 6 10 4 3
2

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。存在的问题主要是，一个是我们递归时的内存管理存在问题，因为我们的 DFS 在实现的时候使用的是递归的方式，且这个递归中有大量的值传递，因此当节点和边的数量非常多的时候有可能会出现栈内存溢出的情况。另一个是我们的最短路径是使用的 BFS，对于有目标搜索使用这种穷举式的搜索是效率较低的，使用启发式搜索则可以提高搜索效率。还有就是因为，为了做这个题目，我们的算法基本都是为了数值类型设计的（int、char、float 等），如果使用其他类型会产生不可预见的 bug，因此可拓展性较差。

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）

```cpp
/*2024 级数据结构--数据智能 实验 12 图 A 图论基础.cpp*/
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

template<typename T>
struct GraphNode
{
    T data;
    //索引方案
    size_t index;
    vector<size_t> neighbors;
    //指针方案
    vector<GraphNode<T>*> neighbors_ptr;
    void addNeighbor(size_t neighbor_index);
    void delNeighbor(size_t neighbor_index);
    void addNeighbor(GraphNode<T>* neighbor_ptr);
    void delNeighbor(GraphNode<T>* neighbor_ptr);
    GraphNode() {}
    GraphNode(T data, size_t index) : data(data), index(index) {}
};

template<typename T>
void GraphNode<T>::addNeighbor(size_t neighbor_index)
{
    for (auto neighbor : neighbors)
```

```
30.         {
31.             if (neighbor == neighbor_index)
32.             {
33.                 return;
34.             }
35.         }
36.         neighbors.push_back(neighbor_index);
37.     }
38.
39.     template<typename T>
40.     void GraphNode<T>::delNeighbor(size_t neighbor_index)
41.     {
42.         for (auto it = neighbors.begin(); it != neighbors.end(); it++)

43.         {
44.             if (*it == neighbor_index)
45.             {
46.                 neighbors.erase(it);
47.                 return;
48.             }
49.         }
50.     }
51.
52.     template<typename T>
53.     void GraphNode<T>::addNeighbor(GraphNode<T>* neighbor_ptr)
54.     {
55.         neighbors_ptr.push_back(neighbor_ptr);
56.     }
57.
58.     template<typename T>
59.     void GraphNode<T>::delNeighbor(GraphNode<T>* neighbor_ptr)
60.     {
61.         neighbors_ptr.erase(find(neighbors_ptr.begin(), neighbors_ptr.e
    nd(), neighbor_ptr));
62.     }
63.
64.     template<typename T>
65.     void printGraphNodeData(GraphNode<T>& node)
66.     {
67.         cout << node.data << " ";
68.     }
69.
70.     template<typename T>
71.     void printGraphNodeEdges(GraphNode<T>& node)
72.     {
73.         cout << "Neighbors: ";
```

```
74.         for (size_t i = 0; i < node.neighbors.size(); i++)
75.         {
76.             cout << node.neighbors[i] + 1 << " ";
77.         }
78.         cout << endl;
79.     }
80.
81.     template<typename T>
82.     void printGraphNode(GraphNode<T>& node)
83.     {
84.         cout << "Node: " << node.data << endl;
85.         printGraphNodeEdges(node);
86.     }
87.
88.
89.
90.     template<typename T>
91.     class GraphWithoutDirection
92.     {
93.     private:
94.         vector<GraphNode<T>> nodes;
95.         T tem_subgraph_data;
96.         void subgraphMinPoints(GraphNode<T>& node);
97.         size_t DFSCount;
98.         void DFSCounter();
99.         size_t BFSCount;
100.        void BFSCounter();
101.    public:
102.
103.        vector<GraphNode<T>>& getNodes() { return nodes; }
104.
105.        GraphWithoutDirection(size_t n);
106.        void addEdge(size_t u, size_t v);
107.        void delEdge(size_t u, size_t v);
108.        void ConnectedComponents();
109.        size_t DFSLength(size_t start_index);
110.        size_t BFSLength(size_t start_index);
111.        void DepthFirstSearch(size_t start_index, vector<bool>& visited
    , void (*operation)(GraphNode<T>& node) = nullptr);//调用类外函数
112.        void DepthFirstSearch(size_t start_index, vector<bool>& visited
    , void (GraphWithoutDirection<T>::* operation)(GraphNode<T>& node));/
    /调用类内函数
113.        void DepthFirstSearch(size_t start_index, vector<bool>& visited
    , void (GraphWithoutDirection<T>::* operation)());//调用类内函数
114.
115.        void DictionaryPrefaceDFS(size_t start_index, vector<bool>& vis
```

```
    ited, void (*operation)(GraphNode<T>& node) = nullptr);//字典序优先深搜+调用类外函数
116.      void DictionaryPrefaceDFS(size_t start_index, vector<bool>& visited, void (GraphWithoutDirection<T>::* operation)(GraphNode<T>& node));//字典序优先深搜+调用类内函数
117.      void DictionaryPrefaceDFS(size_t start_index, vector<bool>& visited, void (GraphWithoutDirection<T>::* operation)());//字典序优先深搜+调用类内函数
118.
119.      void DPDFSPrint(size_t start_index);
120.
121.      void BreadthFirstSearch(size_t start_index, vector<bool>& visited,  void (*operation)(GraphNode<T>& node) = nullptr);//调用类外函数
122.      void BreadthFirstSearch(size_t start_index, vector<bool>& visited,  void (GraphWithoutDirection<T>::* operation)(GraphNode<T>& node));//调用类内函数
123.      void BreadthFirstSearch(size_t start_index, vector<bool>& visited,  void (GraphWithoutDirection<T>::* operation)());//调用类内函数
124.
125.      void DictionaryPrefaceBFS(size_t start_index, vector<bool>& visited,  void (*operation)(GraphNode<T>& node) = nullptr);//字典序优先广搜+调用类外函数
126.      void DictionaryPrefaceBFS(size_t start_index, vector<bool>& visited,  void (GraphWithoutDirection<T>::* operation)(GraphNode<T>& node));//字典序优先广搜+调用类内函数
127.      void DictionaryPrefaceBFS(size_t start_index, vector<bool>& visited,  void (GraphWithoutDirection<T>::* operation)());//字典序优先广搜+调用类内函数
128.
129.      void DPBFSPrint(size_t start_index);
130.
131.      void AllConnectivitySubgraphs();
132.
133.      int Dijkstra(size_t start_index, size_t end_index);
134.  };
135.
136.  template<typename T>
137.  GraphWithoutDirection<T>::GraphWithoutDirection(size_t n)
138.  {
139.      nodes.resize(n);
140.      for (size_t i = 0; i < n; i++)
141.      {
142.          nodes[i].index = i;
143.          nodes[i].data = i + 1;
144.          // nodes[i] = GraphNode<T>(i + 1, i);
145.      }
```

```cpp
}

template<typename T>
void GraphWithoutDirection<T>::addEdge(size_t u, size_t v)
{
    nodes[u - 1].addNeighbor(v - 1);
    nodes[v - 1].addNeighbor(u - 1);
}

template<typename T>
void GraphWithoutDirection<T>::delEdge(size_t u, size_t v)
{
    nodes[u - 1].delNeighbor(v - 1);
    nodes[v - 1].delNeighbor(u - 1);
}

template<typename T>
void GraphWithoutDirection<T>::ConnectedComponents()
{
    size_t count = 0;
    vector<bool> visited(nodes.size(), false);
    for (size_t i = 0; i < nodes.size(); i++)
    {
        if (!visited[i])
        {
            count++;
            visited[i] = true;
            DepthFirstSearch
            (   i,
                visited
                // ,
                // [&](GraphNode<T>& node)
                // {
                //     visited[node.index] = true;
                // }
            );
        }
    }
    // cout << "Connected Components: " << count << endl;
    cout << count << endl;
}

template<typename T>
void GraphWithoutDirection<T>::subgraphMinPoints(GraphNode<T>& node
)
{
```

```
191.     if (node.data < tem_subgraph_data)
192.     {
193.         tem_subgraph_data = node.data;
194.     }
195. }
196. 
197. template<typename T>
198. void GraphWithoutDirection<T>::AllConnectivitySubgraphs()
199. {
200.     vector<T> subgraph_data;
201.     vector<bool> visited(nodes.size(), false);
202.     // auto func =
203.     //     [&min_key](GraphNode<T>& node) mutable -> void
204.     //         {
205.     //             if (node.data < min_key)
206.     //             {
207.     //                 min_key = node.data;
208.     //             }
209.     //     };
210.     // void (*func_ptr)(GraphNode<T>&node) = [](GraphNode<T>& node)
    ;
211.     // func_ptr = func;
212.     for (size_t i = 0; i < nodes.size(); i++)
213.     {
214.         if (!visited[i])
215.         {
216.             tem_subgraph_data = nodes[i].data;
217.             visited[i] = true;
218.             DepthFirstSearch
219.             (
220.                 i,
221.                 visited,
222.                 &GraphWithoutDirection<T>::subgraphMinPoints
223.             );
224.             subgraph_data.push_back(tem_subgraph_data);
225.         }
226.     }
227.     sort(subgraph_data.begin(), subgraph_data.end());
228.     // cout << "All Connectivity Subgraphs: ";
229.     for (auto data : subgraph_data)
230.     {
231.         cout << data << " ";
232.     }
233.     cout << endl;
234. }
235. template<typename T>
```

```
236.  void GraphWithoutDirection<T>::DepthFirstSearch(size_t start_index,
    vector<bool>& visited, void (*operation)(GraphNode<T>& node))
237.  {
238.      for (auto neighbor_index : nodes[start_index].neighbors)
239.      {
240.          if (!visited[neighbor_index])
241.          {
242.              visited[neighbor_index] = true;
243.              if (operation != nullptr)
244.              {
245.                  operation(nodes[neighbor_index]);
246.                  DepthFirstSearch(neighbor_index, visited, operation
   );
247.              }
248.              else
249.              {
250.                  DepthFirstSearch(neighbor_index, visited);
251.              }
252.          }
253.      }
254.  }
255.
256.  template<typename T>
257.  void GraphWithoutDirection<T>::DepthFirstSearch(size_t start_index,
    vector<bool>& visited, void (GraphWithoutDirection<T>::*operation)(G
   raphNode<T>& node))
258.  {
259.      for (auto neighbor_index : nodes[start_index].neighbors)
260.      {
261.          if (!visited[neighbor_index])
262.          {
263.              visited[neighbor_index] = true;
264.              if (operation != nullptr)
265.              {
266.                  (this->*operation)(nodes[neighbor_index]);
267.                  DepthFirstSearch(neighbor_index, visited, operation
   );
268.              }
269.              else
270.              {
271.                  DepthFirstSearch(neighbor_index, visited);
272.              }
273.          }
274.      }
275.  }
276.
```

```
277.  template<typename T>
278.  void GraphWithoutDirection<T>::DepthFirstSearch(size_t start_index,
     vector<bool>& visited, void (GraphWithoutDirection<T>::*operation)()
    )
279.  {
280.      if (operation != nullptr)
281.      {
282.          (this->*operation)();
283.          visited[start_index] = true;
284.      }
285.      for (auto neighbor_index : nodes[start_index].neighbors)
286.      {
287.          if (!visited[neighbor_index])
288.          {
289.              if (operation != nullptr)
290.              {
291.                  DepthFirstSearch(neighbor_index, visited, operation
    );
292.              }
293.              else
294.              {
295.                  DepthFirstSearch(neighbor_index, visited);
296.              }
297.          }
298.      }
299.  }
300.
301.  template<typename T>
302.  void GraphWithoutDirection<T>::DictionaryPrefaceDFS(size_t start_in
    dex, vector<bool>& visited, void (*operation)(GraphNode<T>& node))
303.  {
304.      sort(nodes[start_index].neighbors.begin(), nodes[start_index].n
    eighbors.end());
305.      for (auto neighbor_index : nodes[start_index].neighbors)
306.      {
307.          if (!visited[neighbor_index])
308.          {
309.              visited[neighbor_index] = true;
310.              if (operation != nullptr)
311.              {
312.                  operation(nodes[neighbor_index]);
313.                  DictionaryPrefaceDFS(neighbor_index, visited, opera
    tion);
314.              }
315.              else
316.              {
```

```
                DictionaryPrefaceDFS(neighbor_index, visited);
            }
        }
    }
}

template<typename T>
void GraphWithoutDirection<T>::DictionaryPrefaceDFS(size_t start_index, vector<bool>& visited, void (GraphWithoutDirection<T>::*operation)())
{
    sort(nodes[start_index].neighbors.begin(), nodes[start_index].neighbors.end());
    for (auto neighbor_index : nodes[start_index].neighbors)
    {
        if (!visited[neighbor_index])
        {
            visited[neighbor_index] = true;
            if (operation != nullptr)
            {
                (this->*operation)();
                DictionaryPrefaceDFS(neighbor_index, visited, operation);
            }
            else
            {
                DictionaryPrefaceDFS(neighbor_index, visited);
            }
        }
    }
}

template<typename T>
void GraphWithoutDirection<T>::DictionaryPrefaceDFS(size_t start_index, vector<bool>& visited, void (GraphWithoutDirection<T>::*operation)(GraphNode<T>& node))
{
    sort(nodes[start_index].neighbors.begin(), nodes[start_index].neighbors.end());
    for (auto neighbor_index : nodes[start_index].neighbors)
    {
        if (!visited[neighbor_index])
        {
            visited[neighbor_index] = true;
            if (operation != nullptr)
            {
```

```
356.                     (this->*operation)(nodes[neighbor_index]);
357.                     DictionaryPrefaceDFS(neighbor_index, visited, opera
    tion);
358.                 }
359.                 else
360.                 {
361.                     DictionaryPrefaceDFS(neighbor_index, visited);
362.                 }
363.             }
364.         }
365.     }
366.
367.     template<typename T>
368.     void GraphWithoutDirection<T>::BreadthFirstSearch(size_t start_inde
    x, vector<bool>& visited,  void (*operation)(GraphNode<T>& node))
369.     {
370.         queue<size_t> list;
371.         list.push(start_index);
372.         while (!list.empty())
373.         {
374.             size_t index = list.front();
375.             list.pop();
376.             if (operation != nullptr)
377.             {
378.                 operation(nodes[index]);
379.             }
380.             // sort(nodes[index].neighbors.begin(), nodes[index].neighb
    ors.end());
381.             for (auto neighbor_index : nodes[index].neighbors)
382.             {
383.                 if (!visited[neighbor_index])
384.                 {
385.                     visited[neighbor_index] = true;
386.                     list.push(neighbor_index);
387.                 }
388.             }
389.         }
390.     }
391.
392.     template<typename T>
393.     void GraphWithoutDirection<T>::BreadthFirstSearch(size_t start_inde
    x,vector<bool>& visited,  void (GraphWithoutDirection<T>::*operation)
    ())
394.     {
395.         queue<size_t> list;
396.         list.push(start_index);
```

```
397.     while (!list.empty())
398.     {
399.         size_t index = list.front();
400.         list.pop();
401.         if (operation != nullptr)
402.         {
403.             (this->*operation)();
404.         }
405.          // sort(nodes[index].neighbors.begin(), nodes[index].neighb
   ors.end());
406.         for (auto neighbor_index : nodes[index].neighbors)
407.         {
408.             if (!visited[neighbor_index])
409.             {
410.                 visited[neighbor_index] = true;
411.                 list.push(neighbor_index);
412.             }
413.         }
414.     }
415. }
416.
417. template<typename T>
418. void GraphWithoutDirection<T>::BreadthFirstSearch(size_t start_inde
   x, vector<bool>& visited,  void (GraphWithoutDirection<T>::* operatio
   n)(GraphNode<T>& node))
419. {
420.
421.     queue<size_t> list;
422.     list.push(start_index);
423.     while (!list.empty())
424.     {
425.         size_t index = list.front();
426.         list.pop();
427.         if (operation != nullptr)
428.         {
429.             (this->*operation)(nodes[index]);
430.         }
431.              // sort(nodes[index].neighbors.begin(), nodes[index].ne
   ighbors.end());
432.         for (auto neighbor_index : nodes[index].neighbors)
433.         {
434.             if (!visited[neighbor_index])
435.             {
436.                 visited[neighbor_index] = true;
437.                 list.push(neighbor_index);
438.             }
```

```
439.             }
440.         }
441.     }
442.
443.     template<typename T>
444.     void GraphWithoutDirection<T>::DictionaryPrefaceBFS(size_t start_in
       dex, vector<bool>& visited,  void (*operation)(GraphNode<T>& node))
445.     {
446.         queue<size_t> list;
447.         list.push(start_index);
448.         while (!list.empty())
449.         {
450.             size_t index = list.front();
451.             list.pop();
452.             if (operation != nullptr)
453.             {
454.                 operation(nodes[index]);
455.             }
456.              sort(nodes[index].neighbors.begin(), nodes[index].neighbors
       .end());
457.             for (auto neighbor_index : nodes[index].neighbors)
458.             {
459.                 if (!visited[neighbor_index])
460.                 {
461.                     visited[neighbor_index] = true;
462.                     list.push(neighbor_index);
463.                 }
464.             }
465.         }
466.     }
467.
468.     template<typename T>
469.     void GraphWithoutDirection<T>::DictionaryPrefaceBFS(size_t start_in
       dex, vector<bool>& visited,  void (GraphWithoutDirection<T>::* operat
       ion)())
470.     {
471.         queue<size_t> list;
472.         list.push(start_index);
473.         while (!list.empty())
474.         {
475.             size_t index = list.front();
476.             list.pop();
477.             if (operation != nullptr)
478.             {
479.                 (this->*operation)();
480.             }
```

```
481.          sort(nodes[index].neighbors.begin(), nodes[index].neighbors
   .end());
482.          for (auto neighbor_index : nodes[index].neighbors)
483.          {
484.              if (!visited[neighbor_index])
485.              {
486.                  visited[neighbor_index] = true;
487.                  list.push(neighbor_index);
488.              }
489.          }
490.      }
491.  }
492.
493.
494.  template<typename T>
495.  void GraphWithoutDirection<T>::DictionaryPrefaceBFS(size_t start_in
   dex, vector<bool>& visited,  void (GraphWithoutDirection<T>::* operat
   ion)(GraphNode<T>& node))
496.  {
497.      queue<size_t> list;
498.      list.push(start_index);
499.      while (!list.empty())
500.      {
501.          size_t index = list.front();
502.          list.pop();
503.          if (operation != nullptr)
504.          {
505.              (this->*operation)(nodes[index]);
506.          }
507.          sort(nodes[index].neighbors.begin(), nodes[index].neighbors
   .end());
508.          for (auto neighbor_index : nodes[index].neighbors)
509.          {
510.              if (!visited[neighbor_index])
511.              {
512.                  visited[neighbor_index] = true;
513.                  list.push(neighbor_index);
514.              }
515.          }
516.      }
517.  }
518.
519.  template<typename T>
520.  void GraphWithoutDirection<T>::DPDFSPrint(size_t start_index)
521.  {
522.      // cout << "Dictionary Pre-Order DFS:" << endl;
```

```
523.      vector<bool> visited(nodes.size(), false);
524.      cout << start_index + 1 << " ";
525.      visited[start_index] = true;
526.      DictionaryPrefaceDFS(start_index, visited, printGraphNodeData);

527.      cout << endl;
528.  }
529.
530.  template<typename T>
531.  void GraphWithoutDirection<T>::DPBFSPrint(size_t start_index)
532.  {
533.      // cout << "Dictionary Pre-Order BFS:" << endl;
534.      vector<bool> visited(nodes.size(), false);
535.      // cout << start_index + 1 << " ";
536.      // printGraphNode(nodes[start_index]);
537.      visited[start_index] = true;
538.      DictionaryPrefaceBFS(start_index, visited, printGraphNodeData);

539.      cout << endl;
540.  }
541.
542.  template<typename T>
543.  size_t GraphWithoutDirection<T>::BFSLength(size_t start_index)
544.  {
545.      BFSCount = 0;
546.      vector<bool> visited(nodes.size(), false);
547.      visited[start_index] = true;
548.      BreadthFirstSearch(start_index, visited, &GraphWithoutDirection
    <T>::BFSCounter);
549.      return BFSCount;
550.  }
551.
552.  template<typename T>
553.  void GraphWithoutDirection<T>::BFSCounter()
554.  {
555.      BFSCount++;
556.  }
557.
558.  template<typename T>
559.  size_t GraphWithoutDirection<T>::DFSLength(size_t start_index)
560.  {
561.      DFSCount = 0;
562.      vector<bool> visited(nodes.size(), false);
563.      DepthFirstSearch(start_index, visited, &GraphWithoutDirection<T
    >::DFSCounter);
564.      return DFSCount;
```

```cpp
}

template<typename T>
void GraphWithoutDirection<T>::DFSCounter()
{
    DFSCount++;
}

template<typename T>
int GraphWithoutDirection<T>::Dijkstra(size_t start_index, size_t end_index)
{
    //由于本题目的图是无权图，实际上使用 Dijkstra 算法与使用 BFS 算法的效果是一样的
    //但由于 BFS 算法的实现更简单，所以我们使用 BFS 算法来实现 Dijkstra 算法
    queue<size_t> list;
    vector<int> dist(nodes.size(), 2147483647);
    vector<bool> visited(nodes.size(), false);
    dist[start_index] = 0;
    list.push(start_index);
    while (!list.empty())
    {
        size_t index = list.front();
        list.pop();
        if (!visited[index])
        {
            visited[index] = true;
            for (auto neighbor_index : nodes[index].neighbors)
            {
                list.push(neighbor_index);
                dist[neighbor_index] = min(dist[neighbor_index], dist[index] + 1);
            }
        }

    }
    if (dist[end_index] == 2147483647)
    {
        return -1;
    }
    return dist[end_index];
}


class Solution
```

```
607. {
608. public:
609.     void solve();
610. };
611.
612. void Solution::solve()
613. {
614.     ios::sync_with_stdio(false);
615.     size_t n, m;
616.     cin >> n >> m;
617.     size_t start, end;
618.     cin >> start >> end;
619.     GraphWithoutDirection<int> graph(n);
620.     for (size_t i = 0; i < m; i++)
621.     {
622.         size_t op;
623.         cin >> op;
624.         if (op == 0)
625.         {
626.             size_t u, v;
627.             cin >> u >> v;
628.             graph.addEdge(u, v);
629.         }
630.         else if (op == 1)
631.         {
632.             size_t u, v;
633.             cin >> u >> v;
634.             graph.delEdge(u, v);
635.         }
636.     }
637.     graph.ConnectedComponents();
638.     graph.AllConnectivitySubgraphs();
639.     // cout << "DFS Length: " << graph.DFSLength(start - 1) << endl
    ;
640.     // cout << "BFS Length: " << graph.BFSLength(end - 1) << endl;

641.     // graph.DPDFSPrint(start - 1);
642.     // graph.DPBFSPrint(end - 1);
643.     // cout << "Dijkstra: " << graph.Dijkstra(start - 1, end - 1) <
    < endl;
644.     cout << graph.DFSLength(start - 1) << endl;
645.     graph.DPDFSPrint(start - 1);
646.     cout << graph.BFSLength(end - 1) << endl;
647.     graph.DPBFSPrint(end - 1);
648.     cout << graph.Dijkstra(start - 1, end - 1) << endl;
649.     // for (auto node : graph.getNodes())
```

```
650.     // {
651.     //     printGraphNode(node);
652.     // }
653.     return;
654. }
655. 
656. int main()
657. {
658.     Solution solution;
659.     solution.solve();
660.     return 0;
661. }
```