

**Lab Report**

**实验报告**

|  |  |
| --- | --- |
| **Course**: | Class Libraries and Data Structures |
| **Semester**: | 1st semester of the academic year **2024-2025** |
| **Major**: | Software Engineering |
| **Class**: | 2023 |
| **Student Name**: | 吴仲雄 |
| **Student ID:** | 222023321062007 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Shortest Path Problem on Graphs  图上的最短路径 | | | |
| Date | | Dec，2024 | Type | | ☑Confirmatory （验证确认型）  ☑Design（设计型）  🗆Comprehensive（综合型） |
| 1. **Objective & Requirements（实验目的）**    1. Grasp the adjacency matrix and adjacency list representation method for graph structure   掌握图数据结构的邻接矩阵和邻接链表表示法   * 1. Grasp the basic operations on graph structure   掌握图数据结构的基本操作   * 1. Understand classical algorithms on graphs, such as minimum spanning tree, shortest path, etc.   理解图上常见算法如最小生成树、最短路等的基本原理   * 1. Grasp the implementation of Dijkstra’s algorithm for solving the shortest path problem on graphs.   掌握图上Dijkstra最短路算法的C++实现   * 1. Learn how to use Dijkstra’s algorithm for solving shortest path problems in real applications, for example, path planning on maps.   学习使用最短路算法解决现实应用中的路径规划问题 | | | | | |
| 1. **Experimental environment (**platform and software**)（实验环境）**   Windows 7 (or higher versions) + Visual Studio 2010 (or higher versions) | | | | | |
| 1. Experimental content and design (Main Content, Procedure, Codes and Results)（此部分应包含每一个实验内容的详细设计，含实验思路、详细实验步骤、核心代码说明等）   Task 1  For this task, you are provided with a template container class for graphs. Some basic definitions and operations have already been defined in the class. Try to read and comprehend the codes.  阅读并理解所提供代码对Graph类的实现，特别是其中的邻接链表存储方式。  Based on your understanding of the code, implement the missing method for getting the shortest path from a used provided source vertex v1 and destination vertex v2. The method should return a pair consisting of the vertices on the shortest path, and a double value indicating the path length (in case that there is not a path from v1 to v2, the path is an empty list and the path length is -1.0).  据a), 实现Dijkstra算法，用以求出用户指定的出发点和目的地之间的最短路径以及路径的长度，如果路径不存在，则返回空路径和-1.0.  Hint: it is recommended that a priority queue structure used in the implementation of the get\_shortest\_path() method. Reference:  <https://learn.microsoft.com/en-us/cpp/standard-library/priority-queue-class?view=msvc-170>  <https://en.cppreference.com/w/cpp/container/priority_queue>  <https://cplusplus.com/reference/queue/priority_queue/>  提示：建议在最短路算法中使用优先级队列这种数据结构，可参考上述链接。  Task 2  Using the graph class in Task 1, build a map for SWU campus. The map should consist of at least 20 vertices representing the places on campus that you would visit most often.  使用Task1中的Graph类，创建SWU的校园地图，要求包括至少20个校园地点，地点可参考    The distances between each pair of vertices can be obtained through the measurement service provided by <https://map.baidu.com/>  距离的测量可通过百度地图完成。    Input any two places on the campus map and compute the shortest path between them.  通过SWU校园地图和百度地图测试你对Dijkstra算法的实现。  Task1:  实验思路：  理解并实现图的邻接链表存储方式，并在此基础上完成Dijkstra最短路径算法的实现。Dijkstra算法用于计算从单个源点到其他所有顶点的最短路径，特别适用于边权重非负的情况。本实验要求学生能够根据给定的图结构，计算出用户指定起点v1到终点v2之间的最短路径以及路径长度。  详细实验步骤：   * 阅读与理解：仔细阅读并理解Graph类的现有代码，特别是如何使用邻接链表表示图。 * 分析需求：明确get\_shortest\_path()方法的需求，即返回一个由顶点组成的列表（表示最短路径）和一个double类型的值（表示路径长度），如果不存在路径，则返回空列表和-1.0。 * 设计算法：基于Dijkstra算法原理，设计解决方案，确保算法能正确处理带权图，并且可以找到最短路径。 * 实现优先队列：按照提示，使用C++标准库中的priority\_queue来辅助实现Dijkstra算法。 * 编写代码：根据设计，在Graph类中实现get\_shortest\_path()方法。 * 测试验证：对实现的方法进行充分测试，确保其功能正确性。   核心代码如下：  pair<list<vertex>, double> get\_shortest\_path (const vertex& v1, const vertex& v2)  {  // 使用优先队列来存储待处理的顶点及其权重  priority\_queue<vertex\_weight\_pair, vector<vertex\_weight\_pair>, greater<vertex\_weight\_pair>> preq;  // 存储每个顶点到起点的最短距离  map<vertex, double> distances;  // 存储每个顶点的前驱顶点，用于重建路径  map<vertex, vertex> previous;  // 用于存储最终的最短路径  list<vertex> path;  // 初始化所有顶点的距离为无穷大(使用<limits>库)，前驱顶点为空  for (auto& pair : ver\_adj) {  distances[pair.first] = numeric\_limits<double>::infinity();  previous[pair.first] = vertex();  }  // 起点的距离为0，并将其加入优先队列  distances[v1] = 0.0;  preq.push(vertex\_weight\_pair(v1, 0.0));  // 当优先队列不为空时，处理队列中的顶点  while (!preq.empty()) {  // 取出队列中距离最小的顶点  vertex current = preq.top().to;  preq.pop();  // 如果当前顶点是目标顶点，构建路径并返回  if (current == v2) {  double total\_weight = distances[v2];  for (vertex at = v2; at != vertex(); at = previous[at]) {  path.push\_front(at);  }  return make\_pair(path, total\_weight);  }  // 遍历当前顶点的所有邻接顶点  for (auto& neighbor : ver\_adj[current]) {  // 计算从当前顶点到邻接顶点的距离  double new\_dist = distances[current] + neighbor.weight;  // 如果新的距离小于已知的最短距离，更新距离和前驱顶点，并将邻接顶点加入优先队列  if (new\_dist < distances[neighbor.to]) {  distances[neighbor.to] = new\_dist;  previous[neighbor.to] = current;  preq.push(vertex\_weight\_pair(neighbor.to, new\_dist));  }  }  }  // 如果没有找到路径，返回空路径和-1.0  return make\_pair(list<vertex>(), -1.0);  } // method get\_shortest\_path  源代码(network.h)：  network.h:  #ifndef NETWORK  #define NETWORK  #include <map>  #include <list>  #include <queue>  #include <limits>  using namespace std;  //template<class vertex, class Compare = less<vertex> >  template<class vertex>  class network  {  //node in the adjacency list, a to-vetex, an associated weight  struct vertex\_weight\_pair  {  vertex to;  double weight;  // Postcondition: this vertex\_weight\_pair has been initialized  // from x and y.  vertex\_weight\_pair (const vertex& x, const double& y)  {  to = x;  weight = y;  } // two-parameter constructor  // Postcondition: true has been returned if this  // vertex\_weight\_pair is less than x.  // Otherwise, false has been returned.  bool operator> (const vertex\_weight\_pair& p) const  {  return weight > p.weight;  } // operator>  }; // class vertex\_weight\_pair  typedef typename std::list<vertex\_weight\_pair> adj\_list; //adjacency list  typedef typename adj\_list::iterator adj\_list\_itr; //adjacency list iterotor  typedef typename std::map<vertex, adj\_list> ver\_adj\_map; //map each vetex to its adjacency list  typedef typename ver\_adj\_map::iterator ver\_adj\_map\_itr;  protected:  ver\_adj\_map ver\_adj; //vertex - adjacency list map  public:  // Postcondition: this network is empty.  network() { }  // Postcondition: the number of vertices in this network has been  // returned.  unsigned int size()  {  return ver\_adj.size();  } // method size  // Postcondition: true has been returned if this network contains no  // vertices. Otherwise, false has been returned.  bool empty()  {  return size() == 0;  } // method empty  // Postcondition: true has been returned if this network contains the  // edge <v1, v2>. Otherwise, false has been returned.  bool contains\_edge(const vertex& v1, const vertex& v2)  {  ver\_adj\_map\_itr itr = ver\_adj.find(v1);  if (itr == ver\_adj.end() || ver\_adj.find(v2) == ver\_adj.end()) //not both exist  return false;  adj\_list\_itr list\_itr;  for (list\_itr = ((\*itr).second).begin(); //itr points to a pair in ver\_adj\_map: first:vertex, second:adj\_list  list\_itr != ((\*itr).second).end();  list\_itr++)  if ((\*list\_itr).to == v2) //vertex-weight-struct  return true;  return false;  } // method contains\_edge  // Postcondition: if v is already in this network, false has been  // returned. Otherwise, the map with v and an empty list  // has been added to this network and true has been  // returned.  bool insert\_vertex(const vertex& v)  {  return ver\_adj.insert(  pair<vertex, list<vertex\_weight\_pair> > //map stores pair type  (v, list<vertex\_weight\_pair>()) //empty list  ).second; //map insert returns pair<iterator,bool>    } // method insert\_vertex  // Postcondition: if the edge <v1, v2> is already in this network false  // has been returned. Otherwise, that edge with the  // given weight has been inserted in this network and  // true has been returned.  bool insert\_edge(const vertex& v1, const vertex& v2, const double& weight)  {  if (contains\_edge(v1, v2))  return false;  insert\_vertex(v1); //may already exist  insert\_vertex(v2); //may already exist  (\*(ver\_adj.find(v1))).second.push\_back(vertex\_weight\_pair(v2, weight));  return true;  } // method insert\_edge  /\*=======================================DIJKSTRA'S ALGORITHM==========================================================\*/  // Postcondition: the shortest path from v1 to v2 and its total weight  // have been returned; if such a path is found,  // returns the list of vertices on the shortest path  // and the total weight of the shortest path;  // otherwise, returns an empty list and the value -1.0  pair<list<vertex>, double> get\_shortest\_path (const vertex& v1, const vertex& v2)  {  //please implement this  priority\_queue<vertex\_weight\_pair, vector<vertex\_weight\_pair>, greater<vertex\_weight\_pair>> preq;  map<vertex, double> distances;  map<vertex, vertex> previous;  list<vertex> path;  for (auto& pair : ver\_adj) {  distances[pair.first] = numeric\_limits<double>::infinity();  previous[pair.first] = vertex();  }  distances[v1] = 0.0;  preq.push(vertex\_weight\_pair(v1, 0.0));  while (!preq.empty()) {  vertex current = preq.top().to;  preq.pop();  if (current == v2) {  double total\_weight = distances[v2];  for (vertex at = v2; at != vertex(); at = previous[at]) {  path.push\_front(at);  }  return make\_pair(path, total\_weight);  }  for (auto& neighbor : ver\_adj[current]) {  double new\_dist = distances[current] + neighbor.weight;  if (new\_dist < distances[neighbor.to]) {  distances[neighbor.to] = new\_dist;  previous[neighbor.to] = current;  preq.push(vertex\_weight\_pair(neighbor.to, new\_dist));  }  }  }  return make\_pair(list<vertex>(), -1.0);  } // method get\_shortest\_path  }; // class network  #endif  Task2:  实验思路：  本任务的目标是利用之前实现的get\_shortest\_path()方法来解决实际应用中的路径规划问题。具体来说，就是使用Dijkstra算法求解校园内不同地点之间的最短路径。  详细实验步骤：   * 创建校园地图模型：根据校园内的各个地点及其之间的距离，创建一个图模型。   选择校园内20个常访问的地点，使用百度地图的测量服务获取这些地点之间的直线距离，作为图的边的权重。而后在测试程序中预先插入这些顶点与边。     * 输入处理：通过命令行或GUI界面获取用户输入的起点和终点。 * 调用最短路径函数测试Dijkstra算法：调用之前实现的get\_shortest\_path()方法，传入用户选择的起点和终点，将计算得到的最短路径及其长度输出给用户。 * 多次测试：重复上述过程，对不同的起点和终点进行测试，以确保算法的稳定性和准确性。   核心代码如下(task2.cpp)：  #include <iostream>  #include <string>  #include "network.h"  using namespace std;  int main()  {  network<string> school\_map;  // 插入顶点和边  school\_map.insert\_edge("桃园", "桃园食堂", 121);  school\_map.insert\_edge("桃园食堂", "橘园食堂", 199);  school\_map.insert\_edge("橘园食堂", "第二运动场", 149);  school\_map.insert\_edge("第二运动场", "志成大厦", 217);  school\_map.insert\_edge("志成大厦", "杏园食堂", 51);  school\_map.insert\_edge("杏园食堂", "杏园", 113);  school\_map.insert\_edge("杏园", "北区图书馆", 250);  school\_map.insert\_edge("北区图书馆", "禾丰楼", 300);  school\_map.insert\_edge("禾丰楼", "北区校医院", 100);  school\_map.insert\_edge("北区校医院", "第一运动场", 349);  school\_map.insert\_edge("第一运动场", "出版社", 310);  school\_map.insert\_edge("出版社", "八教", 111);  school\_map.insert\_edge("八教", "李园", 110);  school\_map.insert\_edge("李园", "李园食堂", 220);  school\_map.insert\_edge("李园食堂", "中心图书馆", 200);  school\_map.insert\_edge("中心图书馆", "软件学院", 200);  school\_map.insert\_edge("软件学院", "二十八教", 100);  school\_map.insert\_edge("二十八教", "二十七教", 100);  school\_map.insert\_edge("二十七教", "二十六教", 100);  school\_map.insert\_edge("志成大厦", "二十八教", 303);  school\_map.insert\_edge("志成大厦", "八教", 342);  school\_map.insert\_edge("桃园", "第二运动场", 182);  school\_map.insert\_edge("第二运动场", "禾丰楼", 379);  string start, finish;  cout << "\n请输入起点和终点顶点：" << endl;  cin >> start >> finish;  cout << endl;  pair<list<string>, double> result = school\_map.get\_shortest\_path(start, finish);  list<string>::iterator vertex\_list\_itr;  if (result.first.empty())  cout << "路径不存在！" << endl;  else  {  for (vertex\_list\_itr = result.first.begin();  vertex\_list\_itr != result.first.end();  vertex\_list\_itr++)  cout << \*vertex\_list\_itr << endl;  cout << endl << "从 " << start << " 到 " << finish << " 的最短路径结束" << endl;  cout << "路径权重 = " << result.second << endl;  }  return 0;  } | | | | | |
| 1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）（此部分应包含实验结果，对实验结果的分析，实验收获的总结，实验中存在问题的讨论等；另外，需要回应一下如下思考题：   1) 理论上，Dijkstra算法不能应用于带有负权边的图，为什么？  2) 本次实验所提供代码实现了一个网络类，即带权的有向图，而校园地图通常是无向图，怎么实现用该网络类存储一个无向图？  )  Task1:  实验结果：  输入示例的图进行测试，得到结果：  示例图：    结果：    实验分析：  Dijkstra算法在测试案例中均能正确计算出从起点到终点的最短路径，以及该路径的总权重。测试过程中，通过对比不同节点之间的距离，验证了算法的有效性和准确性。算法的时间复杂度为O((V+E)logV)，其中V是顶点的数量，E是边的数量。对于大多数实际应用来说，这是一个可以接受的效率水平。  实验收获总结：   * 加深了对图论的理解，特别是如何使用邻接表表示图结构。 * 掌握了优先队列的应用，以及它在Dijkstra算法中的作用。 * 提升了编程技能，尤其是处理数据结构和算法的能力。   实验中存在的问题讨论：  在某些情况下，如果图非常大或稠密，Dijkstra算法可能会变得相对慢一些。如果图中有负权边，则需要采用其他算法如Bellman-Ford来解决，因为Dijkstra算法不支持这种情况。代码可能没有充分考虑异常情况，例如输入无效节点名称时的处理。  Task2:  实验结果：        对照测绘图，三组测试数据的最短路径和路径权重均无错误，证明了算法能够准确地应用于实际场景。  实验分析：  测试结果表明，即使是在现实世界的复杂环境中，Dijkstra算法仍然能够有效地找到最短路径。使用C++标准库中的priority\_queue来实现最小堆，提高了查找最短路径的效率。将理论知识应用于实际问题，展示了计算机科学在解决现实问题中的价值。  实验收获总结：   * 学会了如何将理论转化为实践，即如何把课堂上学到的知识应用于真实世界的问题。 * 深入理解了校园内的地理位置关系，并且能够利用这些信息来帮助他人更高效地移动。 * 经历了完整的软件开发生命周期，包括需求分析、设计、编码、测试和评估等阶段。   实验中存在的问题讨论：  虽然Dijkstra算法适用于大部分场景，但对于含有大量节点的地图，可能存在性能瓶颈。校园地图通常是无向图，而实验中使用的网络类默认是有向图。为了适应实际情况，需要确保每条边都以双向形式加入图中。实验未涉及动态更新路径的情况，比如当某些路段临时关闭时，如何快速调整最短路径。需要考虑用户界面友好性，比如提供图形化的地图展示方式，使用户更容易理解和使用系统。  思考题：  1) Dijkstra算法基于贪心策略，即在每一步都选择当前已知的最短路径，并假设这个选择是最终结果的一部分。该算法的工作原理依赖于一个事实：一旦某个节点被确定为拥有从源点出发的最短路径，那么这条路径就不再改变。这是因为Dijkstra算法认为所有边的权重都是非负的，因此后续发现的新路径不会比已经找到的更短。  然而，当图中存在负权边时，上述假设不再成立。具体来说：  如果存在负权环，即一条经过一系列边后回到起始节点且总权重为负数的路径，那么可以无限次地绕行这个环来减少路径长度，导致最短路径问题没有有限解。  即便不存在负权环，负权边也可能使得原本被认为是最短路径的结果变得不是最优。例如，如果通过某条负权边到达的节点后来发现了更短的路径，那么之前确定下来的最短路径就需要更新，而这与Dijkstra算法的工作方式相矛盾。  因此，为了确保正确性，Dijkstra算法不适合处理含有负权边的图。  2) 要将一个带权的有向图网络类转换成适用于表示无向图的数据结构，可以通过以下方法来实现：   * 双向添加边：对于每一个实际存在的无向边，在构建图时需要同时向两个方向插入对应的边。也就是说，如果存在一条连接节点A和节点B的无向边，则应该调用两次insert\_edge()函数，一次是从A指向B，另一次是从B指向A，保证两个方向上的权重相同。这样做的目的是模拟无向图中任意两点之间可以直接互相访问的特点。 * 修改现有接口：如果可能的话，可以在网络类中添加新的接口或方法，专门用于处理无向图的插入操作。比如上面给出的insert\_undirected\_edge()就是一个例子。这样做可以使代码更加清晰易懂，同时也避免了直接修改原有逻辑的风险。   通过上述方法，我们可以有效地利用现有的带权有向图网络类来存储并操作无向图，从而满足校园地图这类应用场景的需求。需要注意的是，这样做可能会增加内存消耗，因为每个实际的无向边在内部被表示为两条独立的有向边。但在大多数情况下，这种额外开销是可以接受的，特别是在节点数量不是特别大的时候。 | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |