

**Lab report**

|  |  |
| --- | --- |
| **Course**: | Class Libraries and Data Structures |
| **Semester**: | 1st semester of the academic year **2021-2022** |
| **Major**: | Software Engineering |
| **Class**: | 2020 |
| **Student Name**: | 温长锟 |
| **Student ID:** | 222020321062106 |
| **Teacher:** | ZHAO, Hengjun (赵恒军) |

**School of Computer and Information Science**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | | Shortest Path Problem on Graphs | | | |
| Date | | Dec，2021 | Type | | □Confirmatory  √ Design  □Comprehensive |
| 1. **Objective & Requirements**    1. Grasp the adjacency matrix and adjacency list representation method for graph structure    2. Grasp the basic operations on graph structure    3. Understand classical algorithms on graphs, such as minimum spanning tree, shortest path, etc.    4. Grasp the implementation of Dijkstra’s algorithm for solving the shortest path problem on graphs.    5. Learn how to use Dijkstra’s algorithm for solving shortest path problems in real applications, for example, path planning on maps. | | | | | |
| 1. **Experimental environment (**platform and software**)**   Windows 7 (or higher versions) + Visual Studio 2010 (or higher versions) | | | | | |
| 1. **Experimental content and design** (Main Content, Procedure, Codes and Results)   Task 1   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. 2. 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 the Boolean flag indicating whether such a path exists (in case that there is not a path from v1 to v2, the Boolean flag is set to false).   Hint: it is recommended that a priority queue structure used in the implementation of the get\_shortest\_path() method. Reference:  <https://www.cplusplus.com/reference/queue/priority_queue/push/>  Task 2   1. 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. (Reference: <http://www.swu.edu.cn/sgis/index.html)>   The distances between each pair of vertices can be obtained through the measurement service provided by <https://map.baidu.com/>   1. Input any two places on the campus map and compute the shortest path between them.   Algorithm features:  Dikoscher algorithm uses breadth first search to solve the single source shortest path problem of weighted directed graph or undirected graph. The algorithm finally obtains a shortest path tree. This algorithm is often used in routing algorithm or as a sub module of other graph algorithms.  Idea of algorithm  Dijkstra algorithm adopts a greedy strategy, declaring an array DIS to save the shortest distance from the source point to each vertex and a set of vertices that have found the shortest path: T. initially, the path weight of the origin s is assigned to 0 (DIS [S] = 0). If there is an edge (s, m) that can be reached directly for vertex s, set dis [M] to w (s, m), and set the path length of all other vertices (which cannot be reached directly by s) to infinity. Initially, set t has only vertex s.  Then, select the minimum value from the dis array, then the value is the shortest path from the source point s to the vertex corresponding to the value, and add the point to t, OK, then a vertex is completed,  Then, we need to see whether the newly added vertex can reach other vertices, and whether the path length to other points through this vertex is shorter than that of the source point. If so, replace the values of these vertices in dis.  Then, find the minimum value from DIS and repeat the above actions until t contains all the vertices of the graph.  #ifndef NETWORK  #define NETWORK  #include <tuple>  #include <map>  #include <list>  #include <queue>  #include <stack>  #include <algorithm>  using namespace std;  //template<class vertex, class Compare = less<vertex> >  template<class vertex, class Compare = less<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, Compare> map\_class; //map each vetex to its adjacency list,map class  typedef typename map\_class::iterator ver\_adj\_map\_itr;  protected:  map\_class adjancency\_map; //vertex -- adjacency map  public:  // Postcondition: this network is empty.  network() { }  // Postcondition: the number of vertices in this network has been  // returned.  unsigned int size()  {  return adjancency\_map.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 = adjancency\_map.find(v1);  if (itr == adjancency\_map.end() || adjancency\_map.find(v2) == adjancency\_map.end()) //not both exist  return false;  adj\_list\_itr list\_itr;  for (list\_itr = ((\*itr).second).begin(); //itr points to a pair in map\_class: 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 adjancency\_map.insert(  pair<vertex, list<vertex\_weight\_pair> > //map stores pair type  (v, list<vertex\_weight\_pair>())).second; //empty list  //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  (\*(adjancency\_map.find(v1))).second.push\_back(vertex\_weight\_pair(v2, weight));  return true;  } // method insert\_edge  /\*=======================================DIJKSTRA'S ALGORITHM==========================================================\*/  class breadth\_first\_iterator  {  friend class network;  protected:  queue<vertex>\* vertex\_queue;  map<vertex, bool, Compare>\* reached;  map\_class\* map\_ptr;//  public:  };  breadth\_first\_iterator(const vertex& start, map\_class\* ptr)  {  map\_ptr = ptr;  reached = new map<vertex, bool, Compare>();  vertex\_queue = new queue<vertex>();  //set the vertex  map\_class::iterator itr;  for (itr = (\*map\_ptr).begin(); itr != (\*map\_ptr).end(); itr++)  (\*reached)[(\*itr).first] == false;  (\*reached)[start] = true;  (\*vertex\_queue).push(start);  }  breadth\_first\_iterator operator++(int)  {  breadth\_first\_iterator temp = \*this;  vertex current = (\*vertex\_queue).front();  (\*vertex\_queue).pop();  map\_class::iterator itr = (\*map\_ptr).find(current);  list<vertex\_weight\_pair>::iterator list\_itr;  for (list\_itr = (\*itr).second.begin(); list\_itr != (\*itr).sencond.end(); list\_itr++)  {  vertex to = (\*list\_itr).to;  if ((\*reached)[to] == false)  {  (\*reached)[to] = true;  (\*vertex\_queue).push(to);  }  }  if ((\*vertex\_queue).empty())  {  vertex\_queue = NULL;  reached = NULL;  map\_ptr = NULL;  }  return temp;  }  breadth\_first\_iterator breadth\_first\_begin(const vertex& v)  {  breadth\_first\_iterator b\_itr(v, &adjancency\_map);  return b\_itr;  }  /\*=======================================DIJKSTRA'S ALGORITHM==========================================================\*/  // Postcondition: the shortest path from v1 to v2 and its total weight  // have been returned.  // returns the list of vertices on the shortest path  // and the total weight of the shortest path  pair<list<vertex>, double> get\_shortest\_path(const vertex& v1, const vertex& v2)  {  const double Max\_PATH\_WEIGHT = 1000000.0;  map<vertex, vertex, Compare> predecessor;  map<vertex, double, Compare>weight\_sum;  priority\_queue<vertex\_weight\_pair, vertex<vertex\_weight\_pair>, greater<vertex\_weight\_pair>>pq;  list<vertex\_weight\_pair>::iterator list\_itr;  breadth\_first\_iterator b\_itr;  vertex to, from;  double weight;  if (adjancency\_map.find(v1) == adjancency\_map.end() || adjancency\_map.find(v2) == adjancency\_map.end())  return pair<list<vertex>, double>(list<vertex>(), -1.0);  bool found\_v2 = false;  for (b\_itr = breadth\_first\_begin(v1); b\_itr != breadth\_first\_end(); b\_itr++)  if (\*b\_itr == v2)  {  found\_v2 = true;  break;  }  if (!found\_v2)  return pair<list<vertex>, double>(list<vertex>(), -1.0);  weight\_sum[v1] = 0.0;  predecessor[v1] = v1;  for (b\_itr = breadth\_first\_begin(v1); b\_itr != breadth\_first\_end(); b\_itr++)  {  weight\_sum[\*b\_itr] = Max\_PATH\_WEIGHT;  predecessor[\*b\_itr] = vertex();  }  for (list\_itr = adjancency\_map[v1].begin(); list\_itr != adjancency\_map[v1].end(); list\_itr++)  {  weight\_sum[list\_itr->to] = list\_itr->weight;  predecessor[list\_itr->to] = v1;  pq.push(vertex\_weight\_pair(\*list\_itr));  }  bool path\_found = false;  while (!path\_found)  {  from = pq.top().to;  pq.pop();  if (from == v2)  path\_found = true;  else  {  for (list\_itr = adjancency\_map[from]; list\_itr != adjancency\_map[from].end(); list\_itr++)  {  to = list\_itr->to;  weight = list\_itr->weight;  if (weight\_sum[from] + weight < weight\_sum[to])  {  weight\_sum[to] = weight\_sum[from] + weight;  predecessor[to] = from;  pq.push(vertex\_weight\_pair(to, weight\_sum[to]));  }  }//  }//no path found  }  list<vertex>path;  vertex current = v2;  while (current != v1)  {  path.push\_front(current);  current = predecessor[current];  }  path.push\_front(v1);  return pair<list<vertex>, double>(path, weight\_sum[v2]);  } // method get\_shortest\_path  }; // class network  #endif  IMG_256  First, in the first step, we declare a dis array, and the initialization value of the array is:  IMG_256  The initialization of our vertex set t is: T = {V1}  Since it is to find the shortest distance from V1 vertex to other vertices, first find the vertex closest to vertex 1. Through the array DIS, we can know that the current vertex closest to V1 is V3 vertex. When vertex 2 is selected, the value of dis [2] (subscript starts from 0) has changed from "estimated value" to "determined value", that is, the shortest distance from V1 vertex to V3 vertex is the current dis [2] value. Add V3 to t.  Why? Because the nearest vertex to V1 is V3 vertex, and all edges of this graph are positive numbers, it is certainly impossible to transit through the third vertex, which further shortens the distance from V1 vertex to V3 vertex. Because the distance from V1 to other vertices is certainly not as short as that from V1 to v3  OK, now that we have determined the shortest path of a vertex, we will find that there are: < V3, V4 > with V3 as the arc tail according to the outgoing degree of the newly entered vertex v3. Let's see whether the length of the path: V1 – V3 – V4 is shorter than V1 – V4. In fact, this is obvious, because dis [3] represents that the length of V1 – V4 is infinite, The length of V1 – V3 – V4 is 10 + 50 = 60, so update the value of dis [3] to obtain the following results:  IMG_256  Therefore, dis [3] will be updated to 60. This process has a technical term called "Relaxation". That is, the distance from V1 vertex to V4 vertex is dis [3], and the edge < V3, V4 > is relaxed successfully. This is the main idea of Dijkstra algorithm: relax the distance from V1 vertex to other vertices through "edge".  Then, we look for the minimum value from other values except dis [2] and dis [0], and find that the value of dis [4] is the smallest. Through the principle explained before, we can know that the shortest distance from V1 to V5 is the value of dis [4]. Then, we add V5 to set t, and then consider whether the output of V5 will affect the value of our array dis. V5 has two outputs: < V5, V4 > and < V5, V6 >, Then we find that the length of V1 – V5 – V4 is 50 and the value of dis [3] is 60, so we need to update the value of dis [3] In addition, the length of v1-v5-v6 is 90 and dis [5] is 100, so we need to update the value of dis [5]. The updated dis array is as follows:  IMG_256  Then, continue to select a minimum value from the values of undetermined vertices in dis. It is found that the value of dis [3] is the smallest, so V4 is added to set T. at this time, set t = {V1, V3, V5, V4}. Then, consider whether the output of V4 will affect the value of our array dis. V4 has an output: < V4, V6 >, and then we find that the length of V1 – V5 – V4 – V6 is 60, The value of dis [5] is 90, so we need to update the value of dis [5]. The updated dis array is as follows:  IMG_256  Then, using the same principle, we determine the shortest paths of V6 and V2 respectively. Finally, the values of the array of dis are as follows:  IMG_256  Therefore, from the figure, we can find that the value of V1-V2 is: ∞, which means that there is no path from V1 to v2. So our final result is:  起点 终点 最短路径 长度  v1 v2 无 ∞  v3 {v1,v3} 10  v4 {v1,v5,v4} 50  v5 {v1,v5} 30  v6 {v1，v5,v4,v6} 60 | | | | | |
| 1. **Result analysis and discussion**（Analysis of experimental results and summing up the harvest and the existing problems）   Under the guidance of Mr. Zhao, I learned a lot in this experiment. This experiment is to find the shortest path.The shortest path problem is a classical algorithm problem in graph theory, which aims to find the shortest path between two nodes in a graph (composed of nodes and paths). The specific forms of the algorithm include:  The shortest path problem of determining the starting point - that is, the problem of finding the shortest path with known starting nodes.  The shortest path problem of determining the end point - contrary to the problem of determining the starting point, this problem is the problem of finding the shortest path with known end nodes. In an undirected graph, this problem is exactly the same as the problem of determining the starting point. In a directed graph, this problem is equivalent to the problem of determining the starting point by reversing all path directions.  The shortest path problem of determining the starting point and end point - that is, the shortest path between two nodes is obtained when the starting point and end point are known.  Global shortest path problem - find all the shortest paths in the graph. | | | | | |
| Comments & Evaluation | Content & Design (A-E) | | |  | |
| Procedure & Codes (A-E) | | |  | |
| Results (A-E) | | |  | |
| Analysis & Discussion (A-E) | | |  | |
| Score (A-E):  Feedback comments: | | | | |