各模块设计报告：

1. 模块调用关系图：
2. 模块功能设计：

2.1 Algorithms模块：

2.1.1 函数设计：

该模块定义了一些函数，这些函数提供了图论中的多种路径搜索算法，包括最短路径和最快路径的查找，以及旅行商问题（TSP）的解决方案。这些函数包括：

1. findMinDistanceNode函数：

（1）功能：该函数用于在未访问的节点中找到具有最小距离的节点。

（2）输入：

·distance：存储节点距离的哈希表。

·visited：存储节点访问状态的哈希表。

·size：节点总数。

（3）处理流程：

a、初始化最小距离为无穷大，未找到的节点编号为-1。

b、遍历所有节点，检查未访问的节点，并更新最小距离和对应的节点编号。

c、返回找到的最小距离节点。

1. *// 辅助函数，用于找到尚未访问的具有最小距离（时间）的节点*
2. int findMinDistanceNode(HashMap<int, double, HashFunc>& distance, HashMap<int, bool, HashFunc>& visited, int size) {
3. double minDistance = std::numeric\_limits<double>::max();
4. int minNode = -1;
5. for (int i = 0; i < size; i++) {
6. double nodeDistance = \*(distance.find(i));
7. if (!\*(visited.find(i)) && nodeDistance < minDistance) {
8. minDistance = nodeDistance;
9. minNode = i;
10. }
11. }
12. return minNode;
13. }
14. findShortestPath函数:

（1）功能：使用 Dijkstra 算法寻找图中两点间的最短路径。

（2）输入：

·graph：图对象。

·startNodeID：起始节点ID。

·endNodeID：目标节点ID。

（3）处理流程：

a、初始化距离、前驱节点和访问标记的哈希表。

b、设置起始节点的距离为0。

c、循环直到找到目标节点或所有节点被访问。

d、使用findMinDistanceNode找到最小距离节点。

e、更新邻居节点的距离和前驱节点。

f、重建路径并返回结果。

1. *// 寻找最短路径算法*
2. Algorithms::PathResult Algorithms::findShortestPath(Graph& graph, int startNodeID, int endNodeID) {
3. int numNodes = graph.size;  *// 图中节点总数*
4. *// 初始化距离、前驱节点和访问标记的哈希表*
5. HashMap<int, double, HashFunc> distances(numNodes);
6. HashMap<int, int, HashFunc> predecessors(numNodes);
7. HashMap<int, bool, HashFunc> visited(numNodes);
8. PathResult result;
9. *// 初始化*
10. for (int i = 0; i < numNodes; ++i) {
11. distances.insert(i, std::numeric\_limits<double>::max());
12. predecessors.insert(i, -1);
13. visited.insert(i, false);
14. }
15. distances[startNodeID] = 0;
16. for (int i = 0; i < numNodes; ++i) {
17. int u = findMinDistanceNode(distances, visited, numNodes);
18. if (u == -1)
19. break;  *// 所有节点都已访问*
20. if (u == endNodeID)
21. break;  *// 找到最短路径*
22. visited[u] = true;
23. for (const auto& edge : graph.getNode(u)->edges) {
24. int v = edge->getTo()->id;
25. double alt = distances[u] + edge->distance;
26. if (alt < distances[v]) {
27. distances[v] = alt;
28. predecessors[v] = u;
29. }
30. }
31. }
32. *// 重建从endNodeId到startNodeId的路径*
33. std::stack<int> pathStack;
34. for (int at = endNodeID; at != -1; at = predecessors[at])
35. pathStack.push(at);
36. if (!pathStack.empty() && pathStack.top() == startNodeID) {  *// 如果路径存在*
37. while (!pathStack.empty()) {
38. result.path.push\_back(pathStack.top());
39. pathStack.pop();
40. }
41. result.length = distances[endNodeID];
42. }
43. return result;
44. }
45. findFastestPath函数：

（1）功能：寻找图中两点间的最快路径。

（2）输入：

·graph：图对象。

·startNodeID：起始节点ID。

·endNodeID：目标节点ID。

·mode：交通方式。

（3）处理流程：

与`findShortestPath`类似，但考虑交通方式和拥堵对时间的影响。

1. *// 寻找最快路径算法*
2. Algorithms::PathResult Algorithms::findFastestPath(Graph& graph, int startNodeID, int endNodeID, int mode) {
3. int numNodes = graph.size;  *// 图中节点总数*
4. *// 初始化时间、前驱节点和访问标记的哈希表*
5. HashMap<int, double, HashFunc> times(numNodes);
6. HashMap<int, int, HashFunc> predecessors(numNodes);
7. HashMap<int, bool, HashFunc> visited(numNodes);
8. PathResult result;
9. for (int i = 0; i < numNodes; i++) {
10. times.insert(i, std::numeric\_limits<double>::max());  *// 初始化时间为无穷大*
11. predecessors.insert(i, -1);                           *// 初始化前驱节点为-1*
12. visited.insert(i, false);                             *// 初始化节点为未访问*
13. }
14. times[startNodeID] = 0;  *// 起点时间设为0*
15. *// 使用Dijkstra算法计算最快路径*
16. for (int i = 0; i < numNodes; i++) {
17. int u = findMinDistanceNode(times, visited, numNodes);  *// 找到未访问的最小时间节点*
18. if (u == -1)                                            *// 无可访问的节点（都已访问）*
19. break;
20. if (u == endNodeID)  *// 如果找到终点，则跳出循环*
21. break;
22. visited[u] = true;  *// 标记该节点为已访问*
23. *// 遍历所有出边，更新时间和前驱节点*
24. int transportmode = static\_cast<Edge::type>(mode);
25. for (const auto& edge : graph.getNode(u)->edges) {
26. if (edge->transportMode == transportmode) {
27. int v = edge->getTo()->id;
28. double alt = times[u] + edge->getLength() / (edge->getSpeed() \* (1 - edge->getCongestion()));
29. if (alt < times[v]) {
30. times[v] = alt;
31. predecessors[v] = u;
32. }
33. } else {
34. *// 如果不是当前交通方式的边，则跳过*
35. continue;
36. }
37. }
38. }
39. *// 从终点开始。使用前驱节点重建路径*
40. std::stack<int> pathStack;
41. for (int at = endNodeID; at != -1; at = predecessors[at])
42. pathStack.push(at);  *// 将节点压入栈*
43. *// 如果路径存在（即栈顶元素为起点），则构建PathResult对象*
44. if (!pathStack.empty() && pathStack.top() == startNodeID) {
45. while (!pathStack.empty()) {
46. result.path.push\_back(pathStack.top());  *// 将栈顶元素添加到路径中*
47. pathStack.pop();                         *// 弹出栈顶元素*
48. }
49. result.time = times[endNodeID];  *// 设置路径的总时间*
50. }
51. return result;  *// 返回路径结果对象*
52. }
53. permutations函数：

（1）功能：生成所有可能的排列，用于 TSP 问题的暴力解法。

（2）输入：

·result：当前最短路径结果。

·graph：图对象。

·startNodeID：起始节点ID。

·arr：节点ID数组。

·l：左侧指针。

·r：右侧指针。

（3）处理流程：

a、如果左右指针相遇，计算路径长度和路径。

b、如果当前路径长度小于已知最短路径，则更新结果。

c、对数组中的每个元素，交换当前元素与左侧元素，递归生成排列。

1. void permutations(Algorithms::PathResult& result, Graph& graph, int startNodeID, std::vector<int> arr, int l, int r) {
2. if (l == r) {
3. *// 基础条件：如果左右指针相遇*
4. double tempLength = 0;
5. std::vector<int> tempPath;
6. for (int i = 0; i < arr.size(); i++) {
7. if (i == 0) {
8. Algorithms::PathResult temp = Algorithms::findShortestPath(graph, startNodeID, arr[i]);
9. tempLength += temp.length;
10. for (int j = 0; j < temp.path.size(); j++) {
11. tempPath.push\_back(temp.path[j]);
12. }
13. } else {
14. Algorithms::PathResult temp = Algorithms::findShortestPath(graph, arr[i - 1], arr[i]);
15. tempLength += temp.length;
16. for (int j = 1; j < temp.path.size(); j++) {
17. tempPath.push\_back(temp.path[j]);
18. }
19. }
20. }
21. Algorithms::PathResult temp = Algorithms::findShortestPath(graph, arr.back(), startNodeID);
22. tempLength += temp.length;
23. for (int j = 1; j < temp.path.size(); j++) {
24. tempPath.push\_back(temp.path[j]);
25. }
26. if (tempLength < result.length) {
27. result.length = tempLength;
28. result.path = tempPath;
29. }
30. } else {
31. for (int i = l; i <= r; i++) {
32. *// 交换 arr[l] 和 arr[i]*
33. std::swap(arr[l], arr[i]);
34. permutations(result, graph, startNodeID, arr, l + 1, r);
35. std::swap(arr[l], arr[i]);
36. }
37. }
38. }
39. findBruteForcePath函数：

（1）功能：使用排列生成所有可能路径，寻找 TSP 问题的解。

（2）输入：

·graph：图对象。

·startNodeID：起始节点ID。

·targets：目标节点集合。

（3）处理流程：

a、初始化结果，设置初始路径长度为无穷大。

b、调用permutations函数生成所有排列并计算路径长度。

c、返回最短路径结果。

1. Algorithms::PathResult Algorithms::findBruteForcePath(Graph& graph, int startNodeID, std::vector<int>& targets) {
2. PathResult result;
3. int size = targets.size();
4. result.length = INF;
5. permutations(result, graph, startNodeID, targets, 0, size - 1);
6. return result;
7. }

2.1.2 数据结构接口：

·Graph：提供节点访问和边遍历的方法。

·HashMap：提供键值对的插入、查找和更新操作。该模块使用哈希表来存储了节点的访问状态、距离、时间等信息。

·PathResult：包含路径长度和路径本身的存储。

2.1.3 算法复杂度分析：

1. findMinDistanceNode：O(n)，其中n是节点总数。

2. findShortestPath和findFastestPath：O(n^2)，在最坏情况下每个节点都可能成为最小距离节点。

3. permutations和findBruteForcePath：O(n!)，因为它们生成所有可能的排列。

2.2 ViewManager模块：

2.2.1 函数设计：

该模块负责管理景点信息，从数据库获取景点数据信息，提供排序和搜索功能。

1. getViews函数：

（1）功能：从数据库中获取景点信息并存储在列表中。

（2）输入：无。

（3）处理流程：

a、初始化数据库连接。

b、连接到数据库。

c、执行查询以获取所有景点信息。

d、遍历查询结果，创建’View’对象，并将其添加到列表中。

（4）getView函数获取查询结果集并添加列表：

1. *// 获取结果集*
2. int count = 0;
3. res = mysql\_store\_result(&my\_sql);
4. while (row = mysql\_fetch\_row(res)) {
5. View temp;
6. temp.LocationID = std::stoi(row[0]);
7. temp.Name = row[1];
8. temp.Type = row[2];
9. temp.Popularity = std::stoi(row[3]);
10. temp.Ratings = std::stod(row[4]);
11. views.push\_back(temp);
12. Recommendation函数：

（1）功能：根据用户的选择推荐景点。

1. 输入：用户依次选择推荐对象（景点、学校或全部）、选择排序方式（热度或评分）、选择推荐数量（前十或全部），以及是否进行关键词搜索。

（3）处理流程：

a、获取用户输入的选择和搜索关键词。

b、根据用户选择调用 ’getScore’ 函数计算综合评分。

c、调用 ‘q\_sort’ 函数对景点列表进行排序。

d、根据用户选择的条件过滤和输出景点信息。

1. */\*排序输出\*/*
2. void ViewManager::Recommendation() {
3. int mode;      *// 0=popularity,1=ratings*
4. int object;    *// 0=attraction,1=school,2=all*
5. int quantity;  *// 0=top10,1=all;*
6. */\*用户进行选择，从而获取三个变量的值\*/*
7. if (mode == 0) {*//热度*
8. if (object == 0) {
9. if (quantity == 0) {
10. */\*推荐热度排名前十位的景区\*/*
11. }else if (quantity == 1) {
12. */\*根据热度对所有景区进行排序推荐\*/*
13. }
14. }else if (object == 1) {
15. if (quantity == 0) {
16. */\*推荐热度排名前十位的学校\*/*
17. }else if (quantity == 1) {
18. */\*根据热度对所有学校进行排序推荐\*/*
19. }
20. }else if (object == 2) {
21. if (quantity == 0) {
22. */\*根据热度对景区和学校的前十位进行推荐\*/*
23. }else if (quantity == 1) {
24. */\*根据热度对所有景区和学校进行排序推荐\*/*
25. }
26. }else if (mode == 1){*//评分*
27. */\*同mode=0时分类方式相同，此处不再赘述\*/*
28. }
29. }
31. getScore函数：

（1）功能：计算景点的综合评分。

（2）输入：评分和热度的权重系数。

（3）处理流程：

a、遍历所有景点。

b、根据给定的权重计算每个景点的综合评分。

c、更新景点对象的评分属性。

1. */\*获取综合评分\*/*
2. void ViewManager::getScore(int a, int b) {
3. int popularity;
4. int ratings;
5. for (int i = 0; i < TEST\_MAXSIZE; i++) {
6. if (views[i].Popularity > MAX\_POPULARITY && b != 0)
7. popularity = 50000;
8. else
9. popularity = views[i].Popularity;
10. ratings = views[i].Ratings \* 100;
11. views[i].Score = a \* popularity + b \* ratings;   *// 计算公式*
12. }
13. }
14. q\_sort函数：

（1）功能：使用快速排序算法对景点列表进行排序。

（2）输入：排序的起始和结束索引。

（3）处理流程：

a、选择一个中间元素作为基准。

b、初始化左右指针进行分区操作。

c、对基准值左边和右边的子数组递归地进行快速排序。

1. */\*排序算法\*/*
2. void ViewManager::q\_sort(int left, int right) {
3. int p = views[(left + right) / 2].Score;
4. int i = left;
5. int j = right;
6. while (i <= j) {
7. while (views[i].Score > p)
8. i++;
9. while (views[j].Score < p)
10. j--;
11. if (i <= j) {
12. std::swap(views[i], views[j]);
13. i++;
14. j--;
15. }
16. }
17. if (j > left)
18. q\_sort(left, j);
19. if (right > i)
20. q\_sort(i, right);
21. }

2.2.2 数据结构接口：

·View：包含景点属性的结构体。

·ViewManager：包含上述函数的类。

2.2.3 算法复杂度分析：

1. getViews：O(n)，其中 n 是数据库中景点的数量。

2. Recommendation：O(nlogn)，由于调用了快速排序。

3. getScore：O(n)，其中 n 是景点数量。

4. q\_sort：O(nlogn)，在最坏情况下的复杂度。

2.3 TestRouter模块：

2.3.1 函数设计：

在复杂的网络中，如校园或城市环境，有效的路径规划对于导航和资源优化至关重要。该模块是用于测试路由算法的核心组件，它集成了图论中的路径搜索算法，并与数据库进行交互以获取网络节点和边的信息。它提供了一个测试和验证这些算法的平台，允许用户选择起点、终点、和途经点并显示搜索结果。

1. RouterTest函数：
2. 功能：

提供了一个测试和验证这些算法的平台，允许用户选择起点、终点、和途经点并显示搜索结果。

（2）输入：无

（3）处理流程；

a、数据库设计、数据查询。

b、图的创建和管理，包含图的初始化、添加节点和边。

c、路径搜索算法的集成：通过与用户交互，选出路线规划方式（1-选择路线规划，2-多个途经点规划）；

如果选择mode=1，那么用户输入起点和终点，调用Algorithms类中的findShortestPath函数计算得到最短路径；

然后用户选择交通方式（1-WALK，2-BIKE），调用Algorithms类中的findFastestPath函数计算得到快路径。

1. *// 使用算法寻找最短路径*
2. Algorithms::PathResult shortestPathResult = Algorithms::findShortestPath(g, from, to);
3. *// 打印结果*
4. std::cout << "Shortest path length: " << shortestPathResult.length << " meters" << std::endl;
5. int flag = 0;
6. for (int nodeID : shortestPathResult.path) {
7. if (flag == 0) {
8. std::cout << nodeID;
9. flag = 1;
10. } else {
11. std::cout << " --> " << nodeID;
12. }
13. }
14. *// 使用算法寻找最快路径*
15. Algorithms::PathResult fastestPathResult = Algorithms::findFastestPath(g, from, to, transport);
16. *// 输出结果*
17. std::cout << "Fastest path time: " << std::fixed << std::setprecision(1) << fastestPathResult.time << " minutes" << std::endl;
18. flag = 0;
19. for (int nodeID : fastestPathResult.path) {
20. if (flag == 0) {
21. std::cout << nodeID;
22. flag = 1;
23. } else {
24. std::cout << " --> " << nodeID;
25. }
26. }

如果选择mode=1，那么用户输入当前位置、以及途经点，调用Algorithms类中的findBruteForcePath函数计算得到最短路径。

最后向用户输出系统所规划出的路线。

1. *// 使用暴力算法*
2. Algorithms::PathResult fspPathResult = Algorithms::findBruteForcePath(g, start, targets);
3. *// 输出结果*
4. std::cout << "Shortest path length: " << fspPathResult.length << " meters" << std::endl;
5. int flag = 0;
6. for (int nodeID : fspPathResult.path) {
7. if (flag == 0) {
8. std::cout << nodeID;
9. flag = 1;
10. } else {
11. std::cout << " --> " << nodeID;
12. }
13. }

2.3.2 数据结构接口：

·Graph g(200)：创建一个具有200个节点容量的图。

·Algorithms::findShortestPath(g, from, to)：使用 Dijkstra 算法寻找图中两点间的最短路径。

·Algorithms::findFastestPath(g, from, to, transport)：寻找图中两点间的最快路径。

·Algorithms::findBruteForcePath(g, start, targets)：查找给定途径点的最短路径。

2.4 TestQuery模块：

2.4.1 函数设计：

TestQuery模块通过与数据库交互，加载地理位置数据，并提供基于位置的查询服务。它支持加载节点和边、执行查询、过滤和排序结果。

1. QueryTest函数：
2. 功能：用于测试和验证基于位置的查询功能，包括从数据库加载数据、执行查询以及过滤和排序结果。
3. 输入：无。
4. 处理流程：

a、数据库设计、数据查询。

b、图的创建和管理，包含图的初始化、添加节点和边：

c、基于位置的查询：

载入场所数据后，通过与用户交互，获取用户的当前位置的节点ID，和查询范围（区域半径），然后调用LocationQuery类中的findNearbyFacilities函数计算得到指定半径范围内的设施，输出第一轮查询结果。

1. std::cout << "Please input the search radius (in meters): ";
2. double searchRadius;
3. std::cin >> searchRadius;
4. *// double searchRadius = 1000;  // 单位为米（或者与Node类位置单位相同）*
5. auto nearbyFacilities = query.findNearbyFacilities(startLocation, searchRadius);
6. query.sortFacilitiesByDistance(nearbyFacilities, 0, nearbyFacilities.size() - 1);
7. *// 输出查询结果*
8. std::cout << "Facilities within " << searchRadius << " m radius:" << std::endl;
9. for (const auto& facility : nearbyFacilities) {
10. std::cout << "Facility Name: " << facility->getName() << "    Distance: " << facility->getDistance() << " meters" << std::endl;
11. }

d、对结果的过滤和排序：

接着用户输入自己感兴趣的设施类别，接下来将按照这个类别，调用LocationQuery类中的filterResultsByCategory函数来按类别对查询结果进行排序（即过滤），调用sortFacilitiesByDistance函数来按距离对设施进行排序，输出查询和排序后的结果。

2.4.2 数据结构接口：

·Graph g(200)：创建一个具有200个节点容量的图。

·LocationQuery auery(q)：创建LocationQuery对象并关联到已有的图。

·query.loadFacilities()：载入场所数据。

·query.findNearbyFacilities(startLocation, searchRadius)：查询指定半径内的设施。

·query.sortFacilitiesByDistance(nearbyFacilities, 0, nearbyFacilities.size() - 1)：按距离对设施进行排序。

·query.filterResultsByCategory(nearbyFacilities, interestedType)：按类别过滤查询结果。

2.5 LocationQuery模块：

2.5.1 函数设计：

LocationQuery模块提供了一套完整的基于位置的查询功能，能够高效地处理信息数据。它支持加载设施数据、查找附近设施、按类别过滤结果和按距离排序结果。

1. LocationQuery函数：

（1）功能：构造一个图。

（2）输入：接受一个“Graph”对象的引用。

（3）处理流程：构造一个图，接受一个图对象的应用，并将其赋值给成员变量graph。

1. LocationQuery::LocationQuery(Graph& g)
2. : graph(g) {}
3. loadFacilities函数：

（1）功能：从图中加载所有设施节点。

（2）输入：无。

（3）处理流程：遍历图中的所有节点，如果节点类型为’Node::Type::FACILITY’，则将其添加到’facilities’向量中。

1. void LocationQuery::loadFacilities() {
2. for (int i = 0; i < graph.size; i++) {
3. Node\* node = graph.getNode(i);
4. if (node->getType() == Node::Type::FACILITY)
5. facilities.push\_back(node);
6. }
7. }
8. findNearbyFacilities函数：

（1）功能：查找给定位置和半径内的设施。

（2）输入：中心地点（圆心）、指定半径。

（3）处理流程：对于’facilities’向量中的每个设施，计算与给定位置的最短路径长度，如果距离小于或等于指定半径，则将该设施添加到结果向量中。

1. std::vector<Node\*> LocationQuery::findNearbyFacilities(Node\* location, double radius) {
2. std::vector<Node\*> nearbyFacilities;
3. for (const auto& facility : facilities) {
4. Algorithms::PathResult pathresult = Algorithms::findShortestPath(graph, location->id, facility->id);
5. double distance = pathresult.length;
6. if (distance <= radius) {
7. facility->setDistance(distance);
8. nearbyFacilities.push\_back(facility);
9. *// distances.push\_back(distance);*
10. }
11. }
12. return nearbyFacilities;
13. }
14. filterResultsByCategory函数：

（1）功能：按照类别过滤设施。

（2）输入：findNearbyFacilities函数查找到的节点向量、类别。

（3）处理流程：遍历结果向量，如果设施的描述与指定类别匹配，则将其添加到过滤后的结果向量中。

1. std::vector<Node\*> LocationQuery::filterResultsByCategory(std::vector<Node\*> results, std::string& category) {
2. std::vector<Node\*> filteredResults;
3. for (const auto& facility : results)
4. if (facility->getDescription() == category)
5. filteredResults.push\_back(facility);
6. return filteredResults;
7. }
8. sortFacilitiesByDistance函数：

（1）功能：按照距离对设施进行排序。

（2）输入：filterResultsByCategory函数过滤出的目标节点向量、两个整数类型的标记（用于快排）。

（3）处理流程：使用快速排序算法根据设施的’getDistance’值进行排序。

1. std::vector<Node\*> LocationQuery::sortFacilitiesByDistance(std::vector<Node\*>& facilities, int low, int high) {
2. if (low < high) {
3. Node\* pivot = facilities[low];  *// 选择第一个元素作为基准*
4. int l = low;
5. int r = high;
6. while (l < r) {
7. while (l < r && facilities[r]->getDistance() >= pivot->getDistance())
8. r--;
9. facilities[l] = facilities[r];
10. while (l < r && facilities[l]->getDistance() <= pivot->getDistance())
11. l++;
12. facilities[r] = facilities[l];
13. }
14. facilities[l] = pivot;
15. *// 分别对分区前后的子数组进行排序*
16. sortFacilitiesByDistance(facilities, low, l - 1);
17. sortFacilitiesByDistance(facilities, r + 1, high);
18. }
19. return facilities;
20. }

2.5.2 数据结构接口：

（1）Graph类

·getNode(int id): 根据节点ID获取节点。

·addNode(int id, Node::Type type, string name, string category): 向图中添加节点。

·addEdge(int start, int end, double distance, double congestion, double speed, Edge::type type): 向图中添加边。

（2）Node类

·getType(): 获取节点类型。

·getDescription(): 获取节点描述。

·setDistance(double distance): 设置节点距离。

（3）Algorithms类

·findShortestPath(Graph& graph, int startNodeID, int endNodeID):寻找图中两点间的最短路径。

2.5.3 算法复杂度分析：

1. loadFacilities()：O(N)，其中N是图中节点的数量。

2. findNearbyFacilities()：O(F \* M)，其中F是设施的数量，M是计算最短路径的时间复杂度（依赖于图的大小和结构）。

3. filterResultsByCategory()： O(F)，其中F是结果向量中设施的数量。

4. sortFacilitiesByDistance(std::vector<Node\*>& facilities, int low, int high)：O(F \* log(F))，其中F是设施的数量。快速排序的平均时间复杂度是O(n \* log(n))。

2.6 Graph模块：

Graph模块定义了一个数据结构，用于表示和管理图中的节点和边。

2.6.1 数据结构设计：

1. 节点（Node）：

节点是图的基本单元，包含以下属性：

·id: 节点的唯一标识符。

·type: 节点的类型。

·name: 节点的名称。

·description: 节点的描述。

2. 边 (Edge)：

边表示节点之间的连接关系，包含以下属性：

·source: 边的起点。

·destination: 边的终点。

·distance: 边的距离。

·congestion: 边的拥堵程度。

·transportMode: 边的交通方式。

·speed: 边的速度。

3. Graph(int size)函数：

（1）功能：构造函数，初始化图的大小，并创建节点哈希表。

（2）输入：一个整数（规定了图的大小、哈希表的大小）。

4. ~Graph(int size)函数：

（1）功能：构造函数，初始化图的大小，并释放所有节点的内存。

（2）输入：无。

5. addNode函数：

（1）功能：添加节点到图中。如果节点ID已经存在，则打印错误信息。

（2）输入：节点id、节点类型、节点的名称、节点的描述信息。

（3）处理流程：判断图中是否有该节点，没找到，则添加该节点；如果可以找到，则说明该节点已经在图中，并打印错误信息。

1. void Graph::addNode(int id, Node::Type type, const std::string& name, const std::string& description) {
2. if (nodes.find(id) == nullptr) {
3. Node\* newNode = new Node(id, type, name, description);
4. nodes.insert(id, newNode);
5. } else {
6. std::cout << nodes.find(id) << std::endl;
7. std::cout << nodes[id]->id << std::endl;
8. *// 节点已存在，打印错误信息*
9. std::cout << "Node with id " << id << " already exists!" << std::endl;
10. }
11. }

6. addEdge函数:

（1）功能：添加边到图中。边是双向的，即如果添加了从节点A到节点B的边，也会自动添加从节点B到节点A的边。

（2）输入：源节点id、目标节点id、边的长度、拥挤度、速度、交通工具种类。

（3）处理流程：调用Graph类中getNode方法，获取对应的源节点、目标节点。检查这两个节点是否都存在，如果任意一节点不存在，函数将输出错误信息并结束。创建从from到to的边和从to到from的边。

1. void Graph::addEdge(const int& from, const int& to, double distance, double congestion, double speed, Edge::type transportMode) {
2. Node\* sourceNode = getNode(from);     *// 获取源节点*
3. Node\* destinationNode = getNode(to);  *// 获取目标节点*
4. if (sourceNode && destinationNode) {
5. *// 添加从from到to的边*
6. Edge\* newEdge = new Edge(sourceNode, destinationNode, distance, congestion, transportMode, speed);
7. sourceNode->addEdge(newEdge);  *// 将该边添加到源节点的边列表中*
8. *// 无向图，添加从to到from的边*
9. Edge\* newEdge2 = new Edge(destinationNode, sourceNode, distance, congestion, transportMode, speed);
10. destinationNode->addEdge(newEdge2);  *// 将该边添加到目标节点的边列表中*
11. } else {
12. std::cout << "Either source or destination node does not exist." << std::endl;
13. }
14. }
15. getNode函数：

（1）功能：根据节点ID检索节点，返回节点指针。

（2）输入：节点id

（3）处理流程：在所有节点中寻找该节点，返回节点id。

1. Node\* Graph::getNode(int id) {
2. return \*(nodes.find(id));
3. }

2.6.2 数据结构接口：

（1）Graph类

·getNode(int id): 根据节点ID获取节点。

·addNode(int id, Node::Type type, string name, string category): 向图中添加节点。

·addEdge(int start, int end, double distance, double congestion, double speed, Edge::type type): 向图中添加边。

（2）Node类

·getType(): 获取节点类型。

·getDescription(): 获取节点描述。

·setDistance(double distance): 设置节点距离。

1. Edge类

·Edge(Node\* from, Node\* to, double distance, double congestion, Edge::type transportMode, double speed)：创建一条边。

2.6.3 算法复杂度分析：

1. addNode方法：O(1)<平均情况，假设哈希表的哈希函数良好>

2. addEdge方法：O(1)<平均情况>

3. getNode方法：O(1)<平均情况>

2.7 Diary模块：

模块成功实现了日记对象的创建、打印和文件写入功能。

2.7.1 模块设计：

1. Diary类：

用于创建和管理日记对象。

2. DiaryPrint成员函数：

将日记的标题、作者、目的地和内容打印到控制台。

3. DiaryWriteintoFile成员函数：

将日记信息写入文件。

4. Diary函数：

（1）功能：初始化日记对象，设置标题、作者、目的地和内容。

（2）输入：标题（std::string title）、作者（std::string author）、目的地（std::string destination）、内容（std::string content）。

（3）处理流程：设置成员变量的值。

5. DiaryPrint函数：

（1）功能：将日记的标题、作者、目的地和内容打印到控制台。

（2）输入：无。

（3）处理流程：格式化输出日记信息和评分。

1. void Diary::DiaryPrint() {
2. std::cout << title << " " << author << " " << destination << "\n";
3. std::cout << content << "\n";
4. std::cout << std::fixed << std::setprecision(1);
5. std::cout << popularity << " " << ((double)rating) / 10000 << "\n";
6. }

6. DiaryWriteintoFile函数：

（1）功能：将日记信息追加写入到指定文件中。

（2）输入：无。

（3）处理流程：

打开文件’D:\\Diarytemp.txt’进行追加写入。检查文件是否成功打开，若失败则输出错误信息并返回。将日记的标题、作者、目的地、内容、热度和评分格式化后写入文件。关闭文件。

1. void Diary::DiaryWriteintoFile() {
2. *// 打开文件进行追加写入*
3. std::string filepath = "D:\\Diarytemp.txt";
4. std::ofstream file(filepath.c\_str(), std::ios::app);
5. if (!file.is\_open()) {
6. std::cerr << "无法打开文件！" << std::endl;
7. return;
8. }
9. *// 将相同的内容写入文件*
10. file << "标题: " << title << '\n';
11. file << "作者: " << author << '\n';
12. file << "游学地点: " << destination << '\n';
13. file << "日记内容: " << content << '\n';
14. file << "热度: " << popularity << '\n';
15. file << "评分: " << ((double)rating) / 10000 << '\n';
16. file << "\n";  *// 添加一个空行作为条目之间的分隔*
17. *// 关闭文件*
18. file.close();
19. }

2.7.2 数据结构接口：

（1）Diary类，包含的属性：

title：日记的标题

author：日记的作者

destination：游学地点

content：日记内容

population：日记的热度

rating：日记的评分

score：日记的得分

2.7.3 算法复杂度分析：

1. Diary::Diary：O(1)。

2. Diary::DiaryPrint：O(n)，其中n是日记内容的长度。

3. Diary::DiaryWriteintoFile：O(m+k)，其中m为日记内容的长度，k为文件操作的常数时间复杂度。

2.8 DiaryManager模块：

模块成功实现了日记管理的各项功能，包括日记的添加、打印、评分、搜索、排序、下载、解压和热度更新。

2.8.1 模块设计：

1. DiaryManager类：

管理日记集合，提供对日记集合的操作。

2. printAllDiaries函数：

（1）功能：打印所有日记的详细信息。

（2）输入：无。

（3）处理流程：

遍历diaries容器中的每个Diary对象。对每个Diary对象调用DiaryPrint方法。3. addDiary函数：

（1）功能：向日记集合中添加一个新的日记。

（2）输入：

（3）处理流程：

使用push\_back方法将传入的diary对象添加到diaries容器中。

4. getScore函数：

（1）功能：根据给定的参数计算每个日记的综合评分。

（2）输入：int a, int b（权重参数）。

（3）处理流程：

遍历 diaries容器，根据 popularity和rating计算每个日记的score。

1. void DiaryManager::getScore(int a, int b) {
2. int popularity;
3. int ratings;
4. for (Diary& diary : diaries) {
5. if (diary.popularity > MAX\_POPULARITY && b != 0)
6. popularity = 50000;
7. else
8. popularity = diary.popularity;
9. ratings = diary.rating;
10. diary.score = a \* popularity + b \* ratings;  *// 计算公式*
11. }
12. }

5. q\_sort函数：

（1）功能：使用快速排序算法对日记集合按照评分进行排序。

（2）输入：int left, int right（排序区间）。

（3）处理流程：

选择中心元素作为基准。通过两两交换的方式将日记分为两部分，一部分的评分高于等于基准，另一部分低于基准。递归地对两部分进行快速排序。

1. void DiaryManager::q\_sort(int left, int right) {
2. int p = diaries[(left + right) / 2].score;
3. int i = left;
4. int j = right;
5. while (i <= j) {
6. while (diaries[i].score > p)
7. i++;
8. while (diaries[j].score < p)
9. j--;
10. if (i <= j) {
11. std::swap(diaries[i], diaries[j]);
12. i++;
13. j--;
14. }
15. }
16. if (j > left)
17. q\_sort(left, j);
18. if (right > i)
19. q\_sort(i, right);
20. }

6. diarySearch函数：

（1）功能：根据标题、作者、目的地和内容搜索日记，并按指定模式排序。

（2）输入：

std::string search\_title, search\_author, search\_destination, search\_content, int search\_mode。

（3）处理流程：

根据search\_mode计算评分并排序日记。清空缓存文件。遍历日记集合，使用 KMP 算法匹配搜索条件。匹配成功的日记将被打印和写入缓存文件。

1. void DiaryManager::diarySearch(std::string search\_title, std::string search\_author, std::string search\_destination, std::string search\_content, int search\_mode) {
2. if (search\_title == "-1")
3. search\_title = "";
4. if (search\_author == "-1")
5. search\_author = "";
6. if (search\_destination == "-1")
7. search\_destination = "";
8. if (search\_content == "-1")
9. search\_content = "";
10. int a;
11. int b;
12. if (search\_mode == 0)  *// 按popularity*
13. {
14. a = 1;
15. b = 0;
16. } else if (search\_mode == 1)  *// 按rating*
17. {
18. a = 0;
19. b = 1;
20. }
21. getScore(a, b);
22. q\_sort(0, diaries.size() - 1);
23. *// 清空下载缓存文件*
24. std::string filepath = "D:\\Diarytemp.txt";
25. std::ofstream file(filepath.c\_str());
26. if (!file.is\_open()) {
27. std::cerr << "无法打开缓存文件！" << std::endl;
28. }
29. file.close();
30. *// 匹配操作*
31. for (Diary diary : diaries) {
32. if (kmp(search\_title, diary.title) && kmp(search\_author, diary.author) && kmp(search\_destination, diary.destination) && kmp(search\_content, diary.content)) {
33. diary.DiaryPrint();
34. diary.DiaryWriteintoFile();
35. }
36. }
37. }

7. diaryDownload函数：

（1）功能：下载日记集合。

（2）输入：无。

（3）处理流程：调用FileCompress类的Compress方法压缩日记文件。

8. diaryUncompress函数：

（1）功能：解压日记文件。

（2）输入：std::string path（文件路径）

（3）处理流程：调用FileCompress类的UnCompress方法解压文件。

9. up\_popularity函数：

（1）功能：根据日记内容更新日记的热度。

（2）输入：std::string content

（3）处理流程：

遍历日记集合，使用 KMP 算法匹配内容。匹配成功的日记热度增加。

1. *// 热度自增*
2. int DiaryManager::up\_popularity(std::string content) {
3. for (Diary diary : diaries) {
4. if (kmp(content, diary.content)) {
5. diary.popularity += 10;
6. return diary.popularity;
7. }
8. }
9. return -1;
10. }

10. update\_rate函数：

（1）功能：更新指定日记的评分。

（2）输入：std::string content, int new\_rating

（3）处理流程：

遍历日记集合，使用 KMP 算法匹配内容。匹配成功的日记评分更新。

1. *// 日记评分*
2. int DiaryManager::update\_rate(std::string content, int new\_rating) {
3. for (Diary diary : diaries) {
4. if (kmp(content, diary.content)) {
5. diary.rating = (diary.rating + new\_rating) / 2;
6. return diary.rating;
7. }
8. }
9. return -1;
10. }

2.8.2 数据结构接口：

（1）Diary类

（2）DiaryManager类

2.8.3 算法复杂度分析：

1. printAllDiaries()：O(n\*m)，其中n是日记的数量，m是打印一个日记所需的时间（与日记内容长度相关）。

2. addDiary(Diary diary)：O(1)，因为 std::vector的push\_back操作通常是常数时间的。

3. getScore(int a, int b)：O(n)，其中n是日记的数量，因为需要遍历一次所有日记。

4. q\_sort(int left, int right)：O(n log n)，在最坏情况下，快速排序的时间复杂度为O(n^2)，但平均情况下是O(n log n)。

5. diarySearch()：对于n个日记，总复杂度为O(n\*(m+k))其中m是搜索字符串的长度，k是日记内容的长度。

6. diaryUncompress(std::string path)：取决于FileCompress类的UnCompress方法实现，解压操作的时间复杂度通常与压缩数据的大小和算法有关。

7. up\_popularity(std::string content)：O(n\*k)，其中n是日记的数量，k是内容字符串的长度。这里假设 `kmp` 函数在最坏情况下的时间复杂度为O(k)。

8. update\_rate(std::string content, int new\_rating)：O(n\*k)。

2.9 DiarySearchDown模块：

模块成功执行了以下操作：

- 数据库连接和查询。

- 日记的添加、搜索、排序、下载和解压。

- 日记热度和评分的更新。

2.9.1 模块设计：

1. DiaryTest函数：

（1）功能：根据模式执行不同的日记管理操作。

（2）输入：操作模式、日记的属性、排序方式或新评分。

（3）处理流程：

a. 初始化和连接MySQL数据库。

b. 查询数据库中的日记，并创建’Diary’对象。

c. 根据’mode’执行相应操作：

- 模式0：执行搜索和排序测试。

- 模式1：执行日记下载。

- 模式2：执行日记解压。

- 模式3：更新日记热度并更新数据库。

- 模式4：更新日记评分并更新数据库。

- 其他模式：将新日记写入数据库。

d. 关闭数据库连接。

2.9.2 数据结构接口：

（1）Diary类

（2）DiaryManager类

（3）MYSQL数据库

2.9.3 算法复杂度分析：

1. DiaryTest函数:

该集成了数据库操作和’DiaryManager’的多个功能，其整体复杂度受到数据库大小、日记集合大小和执行的操作类型的影响。数据库操作通常涉及磁盘I/O，可能会成为性能瓶颈。’DiaryManager’的操作，如搜索和排序，具有较高的时间复杂度，但它们是在内存中对相对较小的数据集进行操作，因此通常比数据库操作更快。

2.10 Facility模块：

Facility 模块定义了一个设施（Facility）的类，这个类用于表示一个具体的设施，包括其位置、名称和类型。该模块主要用于管理和提供设施的基本信息。

2.10.1 模块设计：

Facility类包含三个私有成员变量：

1. location：一个Node类型的对象，表示设施的位置。
2. name：一个std::string类型的变量，表示设施的名称。
3. type：一个std::string类型的变量，表示设施的类型。

此外，该类还提供了以下公共成员函数：

1. 构造函数Facility(const Node& loc, const std::string& name, const std::string& type)：用于初始化设施对象。
2. 成员函数 std::string getType() const：用于获取设施的类型。
3. 成员函数 std::string getType() const：用于获取设施的类型。
4. 成员函数const Node& getLocation() const：用于获取设施的位置。

2.11 FileCompress函数：

FileCompress 模块提供文件压缩和解压缩的功能。它使用哈夫曼编码算法对文件进行压缩，将文件内容压缩成一个 .zlx 文件，并提供解压缩功能，将 .zlx 文件恢复成原始文件

2.11.1 模块设计：

FileCompress 类包含多个私有成员函数和数据结构来实现文件的压缩和解压缩。关键成员包括：

1. info 数组：存储字符及其出现次数和编码信息的数组。
2. \_SIZE\_ 常量：定义了读取和写入缓冲区的大小。
3. \_FILE\_NAME\_SIZE\_ 常量：定义了文件名大小的限制。

该类提供的主要公共成员函数包括：

1. Compress(const std::string& FilePath)：压缩指定路径的文件。
2. UnCompress(const std::string& FilePath)：解压缩指定路径的文件。

2.11.2 各个函数功能及其处理流程：

1. Compress(const std::string& FilePath)：

（1）功能：压缩指定路径的文件。

（2）输入：待处理文件路径。

（3）处理流程：

a、打开输入文件；

b、调用 FillInfo 函数填充字符出现次数信息；

c、获取压缩后的文件名，并打开输出文件；

d、调用 CompressCore 函数执行压缩核心逻辑；

e、 关闭文件。

2. UnCompress(const std::string& FilePath)函数：

（1）功能：解压缩指定路径的文件

（2）输入：文件路径。

（3）处理流程：

a、打开输入文件；

b、调用“GetHead”函数读取压缩文件头部信息；

c、创建输出文件；

d、初始化哈夫曼树；

e、调用“UnCompressCore”函数执行解压缩核心逻辑；

f、关闭文件。

3. UnCompressCore函数：

（1）功能：解压缩核心逻辑；

（2）输入：输入数据、输出数据、哈夫曼树结构。

（3）处理流程：

a、初始化读取和写入缓冲区；

b、从输入文件中读取压缩数据；

c、根据哈夫曼解码数据并写入输出文件。

4. GetHead函数：

（1）功能：获取压缩文件头部信息。

（2）输入：文件后缀名。

（3）处理流程：

a、获取文件后缀名；

b、获取编码行数；

c、获取并设置每个字符的出现次数。

5. GetLine函数：

（1）功能：从文件中读取一行。

（2）输入：文件、缓冲区大小。

（3）处理流程：

从输入文件中逐字节读取，直到读取到换行符或达到缓冲区大小。

6.GetFileName函数：

（1）功能：从文件路径中提取文件名（不含路径和后缀）。

（2）输入：文件路径、输出。

（3）处理流程：

a、找到最后一个路径分隔符的位置；

b、找到最后一个点号的位置；

c、提取文件名部分。

7. GetPostfixName函数：

（1）功能：从文件路径中提取后缀名。

（2）输入：文件路径、输出。

（3）处理流程：

a、找到最后一个点号的位置；

b、提取后缀名部分。

8. Fillnfo函数：

（1）功能：填充字符出现次数信息并生成哈夫曼编码。

（2）输入：

（3）处理流程：

a、初始化 info 数组；

b、统计每个字符在文件中出现的次数；

c、构建哈夫曼树；

d、生成每个字符的哈夫曼编码。

9. FillCode函数：

（1）功能：递归生成哈夫曼编码。

（2）输入：

（3）处理流程：

a、递归遍历哈夫曼树；

b、对于叶子节点，生成对应的编码并存储在 info 数组中。

10. CompressCore函数：

（1）功能：压缩核心功能。

（2）输入：待处理文件、文件路径。

（3）处理流程：

a、将输入文件的指针重置到文件开头；

b、保存压缩文件的编码头信息；

c、读取输入文件并进行编码转换，逐字节写入输出文件。

11. SaveCode函数：

（1）功能：保存压缩文件的编码头信息。

（2）输入：目的文件、文件路径。

（3）处理流程：

a、获取文件后缀名并写入输出文件；

b、写入编码行数；

c、写入每个字符及其出现次数。

2.11.2 算法复杂度分析：

1. Compress(const std::string& FilePath)：**总时间复杂度**：O(m) + O(n) + O(c) ≈ O(m + n + c) （其中 m 为文件大小，n 为文件路径长度，c 为字符编码数）。

2. UnCompress：**总时间复杂度**：O(h) + O(n) + O(u) ≈ O(h + n + u) （其中 h 为头部信息的大小，n 为文件路径长度，u 为文件大小）。

3. UnCompressCore(FILE\* input, FILE\* output, HuffmanTreeNode<CodeInfo>\* pRoot)：**总时间复杂度**：O(m) （m 为文件大小）。

4. FillInfo(FILE\* src)：**总时间复杂度**：O(m) + O(c log c) ≈ O(m + c log c)。

2.12 Kmp模块：

该模块实现了 KMP（Knuth-Morris-Pratt）字符串匹配算法，用于在一个字符串中查找另一个字符串的出现位置。

2.12.1 模块设计：

1. get\_nextval函数：

（1）功能：计算模式字符串 t 的 next 数组，用于 KMP 算法。

（2）输入：待处理字符串，nextval数组。

（3）处理流程：

a、初始化变量 j 为 0，k 为 -1；

b、设置 nextval[0] 为 -1；

1. void get\_nextval(std::string t, int nextval[]) {
2. int j = 0, k = -1;
3. int t\_len = t.length();
4. nextval[0] = -1;
5. while (j < t\_len)
6. if (k == -1 || t[j] == t[k]) {
7. j++;
8. k++;
9. if (t[j] != t[k])
10. nextval[j] = k;
11. else
12. nextval[j] = nextval[k];
13. } else
14. k = nextval[k];
15. }

2.12.2 kmp函数：

（1）功能：在目标字符串 s 中查找模式字符串 t。

（2）输入：目标字符串s，模式字符串t。

（3）处理流程：

a、如果 t 等于 "-1"，返回 true；

b、初始化 nextval 数组，长度为 t 的长度；

c、调用 get\_nextval 函数计算模式字符串 t 的 nextval 数组；

d、初始化变量 i 和 j 为 0；

e、进入 while 循环；

f、如果 j 大于或等于模式字符串长度，返回 true，表示匹配成功；否则返回 false。

1. bool kmp(std::string t, std::string s) {
2. if (t == "-1")
3. return 1;
4. int line\_limit = t.length();
5. int nextval[line\_limit];
6. int i = 0, j = 0;
7. int s\_len = s.length(), t\_len = t.length();
8. get\_nextval(t, nextval);
9. while (i < s\_len && j < t\_len)
10. if (j == -1 || s[i] == t[j]) {
11. i++;
12. j++;
13. } else
14. j = nextval[j];
15. if (j >= t\_len)
16. return true;
17. else
18. return false;
19. }

2.13 MyHashMap函数：

这个模块实现了一个模板化的哈希表（HashMap），用于存储键值对。它支持插入、删除、查找、以及通过索引访问和打印整个哈希表。

2.13.1 模块设计：

1.insert函数：

（1）功能：向哈希表中插入一个键值对。

（2）输入：一个键值对。

（3）处理流程：

a、计算键的哈希值，并取模计算槽位索引；

b、创建一个新节点，存储键值对；

c、将新节点插入到对应槽位的链表头部。

1. template <typename Key, typename Value, typename HashFunc>
2. bool HashMap<Key, Value, HashFunc>::insert(Key key, Value value) {
3. int index = hash(key) % size;
4. HashNode<Key, Value>\* newnode = new HashNode<Key, Value>(key, value);
5. newnode->next = table[index];
6. table[index] = newnode;
7. return true;
8. }

2. del函数：

（1）功能：从哈希表中删除指定键的节点。

（2）输入：一个键。

（3）处理流程：

a、计算键的哈希值，并取模计算槽位索引；

b、遍历对应槽位的链表，查找匹配的节点；

c、如果找到匹配节点，将其从链表中移除并删除。

1. *// 删除函数*
2. template <typename Key, typename Value, typename HashFunc>
3. bool HashMap<Key, Value, HashFunc>::del(Key& key) {
4. int index = hash(key) % size;
5. HashNode<Key, Value>\* cur = table[index];
6. HashNode<Key, Value>\* pre = nullptr;
7. while (cur != nullptr) {
8. if (cur->key == key) {
9. if (pre != nullptr) {
10. pre->next = cur->next;
11. } else {
12. table[index] = cur->next;
13. }
14. delete cur;
15. return true;
16. }
17. pre = cur;
18. cur = cur->next;
19. }
20. return false;
21. }
22. find函数：

（1）功能：查找并返回指定键的值。

（2）输入：一个键。

（3）处理流程：

a、计算键的哈希值，并取模计算槽位索引；

b、遍历对应槽位的链表，查找匹配的节点；

c、如果找到匹配节点，返回其值的指针，否则返回 nullptr。

1. template <typename Key, typename Value, typename HashFunc>
2. Value\* HashMap<Key, Value, HashFunc>::find(Key& key) {
3. int index = hash(key) % size;
4. HashNode<Key, Value>\* cur = table[index];
5. while (cur != nullptr) {
6. if (cur->key == key)
7. return &(cur->value);
8. cur = cur->next;
9. }
10. return nullptr;
11. }

2.14 Edge模块：

这个模块实现了 Edge 类，用于表示图中的边。Edge 类包含源节点、目标节点、距离、拥堵情况、交通方式和速度等属性，并提供了一些获取这些属性的方法。

2.14.1 模块设计：

1. 成员变量：

· Node\* source: 边的起点节点

· Node\* destination: 边的终点节点

· double distance: 边的长度（距离）

· double congestion: 边的拥堵情况

· type transportMode: 边的交通方式

· double speed: 边的速度

2. 成员函数：

· 构造函数 Edge

· 获取源节点 getFrom

· 获取目标节点 getTo

· 获取距离 getLength

· 获取拥堵情况 getCongestion

· 获取速度 getSpeed

· 获取交通方式 gettype

2.15 Node模块：

这个模块实现了 Edge 类，用于表示图中的边。Edge 类包含源节点、目标节点、距离、拥堵情况、交通方式和速度等属性，并提供了一些获取这些属性的方法。

2.15.1 模块设计：

1. 成员变量：

· int id: 节点的唯一标识符

· Type type: 节点的类型（枚举类型）

· std::string name: 节点的名称

· std::string description: 节点的描述

· std::vector<Edge\*> edges: 与节点相连的边

· double distance: 节点距离（用于路径算法）

2. 成员函数：

· 构造函数 Node

· 析构函数 ~Node

· 获取节点 ID getId

· 获取节点名称 getName

· 获取节点类型 getType

· 获取节点描述 getDescription

· 设置节点描述 setDescription

· 获取节点距离 getDistance

· 设置节点距离 setDistance

· 添加边 addEdge

· 静态方法 emptyNode 返回一个空节点对象