# PJ 实验报告

姓名：涂睿睿

学号：23300240031

## 实验要求

​ 本次项目的目标是将C++ 后端的地图数据处理和路径计算功能与 JavaScript 前端的交互和地图渲染结合实现一个基于地图数据的路径规划系统。

​ 我在基本的要求的基础上还另外实现了（1）支持区分行人和车辆模式、（2）支持地图上任一点的选取。最后系统通过读取地图数据绘制了节点、道路信息和建筑物，在用户点击地图上任意两点后基于所选择模式与上述数据计算从起点到终点的最短路径并显示。

​ 在初步完成地图路径规划项目以后，课程设置了pj展示。再观看了其他同学的pj演示以后，我又在原本所完成的基础上通过调用百度地图的api进一步丰富了我的前端实现。同时，由于百度地图的前端已经提供了整个中国的数据，所以升级后的数据量也扩大了。

## 实现思路

​ 本次项目的实现思路主要可以分为以下几个关键部分：数据获取、路径计算、前后端交互、前端界面以及性能考虑和最终扩展。

### 数据获取

​ 本次项目所需要的地图数据均从OSM（OpenStreetMap）网站获取，以 **XML** 格式的地图数据下载至本地。本次地图数据我选择了我的家（上海宝山）附近一带的地图数据。这些数据包含了地图的节点（如 GPS 坐标）、道路（如连接节点的路径）以及建筑物信息（封闭回路的way）。具体步骤如下：

* **加载地图数据**：通过 TinyXML 解析器从 XML 文件中加载地图数据，并提取其中的节点和道路信息。每个节点有一个唯一的 ID，并包含经纬度信息。
* **构建邻接图**：通过解析每条道路的节点连接关系，构建 **邻接图**，该图将每个节点与它相邻的节点连接起来。为了支持行人和车辆两种不同的交通模式，我分别构建了两个邻接图：pedestrianAdjacencyMap（行人模式）和 vehicleAdjacencyMap（车辆模式）。

这两种模式的区分在后端依靠isPedestrianAccessible函数和isVehicleAccessible函数完成。

1. bool isPedestrianAccessible(const Json::Value& tags) {

2. if (tags.empty()) return true; // Default to accessible if no tags

3.

4. string highway = tags.get("highway", "").asString();

5. string footway = tags.get("footway", "").asString();

6. string pedestrian = tags.get("pedestrian", "").asString();

7. string access = tags.get("access", "").asString();

8.

9. // Pedestrian-specific paths

10. if (footway == "yes" || pedestrian == "yes" ||

11. highway == "footway" || highway == "pedestrian" ||

12. highway == "path" || highway == "steps" ||

13. highway == "crossing" || highway == "sidewalk") {

14. return true;

15. }

16.

17. // Pedestrian-allowed in mixed traffic zones

18. if ((highway == "residential" || highway == "living\_street" ||

19. highway == "service" || highway == "track") &&

20. access != "no" && access != "private") {

21. return true;

22. }

23.

24. // Further check for restricted areas (if specified by tags)

25. if (tags.isMember("no\_exit") || tags.isMember("dead\_end")) {

26. return false;

27. }

28.

29. return false;

30. }

31. bool isVehicleAccessible(const Json::Value& tags) {

32. if (tags.empty()) return true; // Default to accessible if no tags

33.

34. string highway = tags.get("highway", "").asString();

35. string motor\_vehicle = tags.get("motor\_vehicle", "").asString();

36. string access = tags.get("access", "").asString();

37.

38. // Check if motor vehicle access is explicitly denied

39. if (motor\_vehicle == "no" || access == "no" || access == "private") return false;

40.

41. // Vehicle-specific roads (public roads typically accessible to vehicles)

42. if (highway == "motorway" || highway == "trunk" ||

43. highway == "primary" || highway == "secondary" ||

44. highway == "tertiary" || highway == "residential" ||

45. highway == "service" || highway == "unclassified" ||

46. highway == "living\_street") {

47. return true;

48. }

49.

50. // Allow vehicle access if road type is not specifically restricted

51. if (highway == "track" && motor\_vehicle != "no") {

52. return true;

53. }

54.

55. // Further check for restricted areas like private roads or areas marked for certain vehicle types

56. if (tags.isMember("access") && (access == "destination" || access == "customer")) {

57. return true;

58. }

59.

60. return false;

61. }

62.

其中，关于如何根据道路的节点来判断行人/车辆能否通行，我的主要依据是[OpenStreetMap Wiki上有关于highway的解释](https://wiki.openstreetmap.org/wiki/Key:highway)。

* **道路标签分析**：每条道路都有一组标签，用于标识该道路是否适合行人或车辆通行。例如，如果某条道路标记为 "highway=footway"，则表示该道路适合行人通行。通过分析这些标签，我们可以决定某条道路是否应被添加到行人或车辆的邻接图中。

### 路径计算

路径计算是本次项目的核心功能，具体为后端文件中的findShortestPath函数，目标是根据用户选择的起点和终点，计算最短路径。为了实现这一功能，我使用了 **Dijkstra 算法的堆优化方法**。具体细节如下：

* **Dijkstra 算法**：Dijkstra 算法 用于计算从起点到终点的最短路径。为了提高算法的效率，本次项目采用了 **堆优化**（即优先队列）来加速 Dijkstra 算法的执行过程。
  1. **Dijkstra 算法基本思想**

Dijkstra 算法的核心思想是通过贪心策略，逐步寻找当前节点到源节点的最短路径。在每一次迭代中，它选择当前已知最短路径的节点，并扩展到它的邻居节点，直到遍历完所有可达节点或找到目标节点

其步骤如下：

* + 1. 初始化：每个节点的距离设置为正无穷大，起点的距离设置为 0。
    2. 使用优先队列（堆）来存储和处理当前最短路径已知的节点。每次从队列中取出距离最小的节点，更新该节点的邻居的最短距离。
    3. 重复步骤 2，直到队列为空或者目标节点的最短路径被确定。
  1. **堆优化的引入**

​ 堆是一个可以高效获取最小（或最大）元素的数据结构。在 Dijkstra 算法中，堆用于动态获取当前已知的最短路径节点。本次项目使用 最小堆来存储所有节点和它们的当前最短距离，堆的根节点总是距离最短的节点。每次从堆中弹出距离最短的节点，并更新邻居的最短距离。

​ 堆优化使得 Dijkstra 算法的时间复杂度从原来的 **O(V²)**（如果用数组存储距离）或者 **O(VE)**（如果使用邻接表）降低到了 **O((V + E) log V)**。

* 1. 在本次代码中该算法的**具体实现**如下：
     1. 优先队列（堆）的初始化

在 Dijkstra 算法中，堆优化的核心是使用 优先队列 来存储和获取当前已知的最短路径节点。这里使用了 C++ 标准库中的 priority\_queue，它是一个默认的最大堆（大根堆），但通过自定义比较器 greater<>()，将其转化为最小堆，从而保证每次弹出的是最短距离的节点。

1. priority\_queue<pair<double, string>, vector<pair<double, string>>, greater<>> pq;

* + - * pair<double, string>：pair 中的 double 表示当前节点的最短距离，string 表示节点的 ID。
      * greater<>：C++ 中的 greater 比较器，用来构建最小堆。默认情况下，priority\_queue 是一个最大堆（最大元素在堆顶），greater<>() 将堆按升序排列，从而使得每次弹出的元素是当前最小的元素。
    1. 初始化距离数组

在算法开始时，所有节点的最短路径距离初始化为无穷大，表示未知距离。只有起点的距离初始化为 0。然后将起点加入优先队列中。

1. map<string, double> distances;

2. map<string, string> previous;

3. // 初始化所有节点的距离为无穷大

4. for (const auto& node : nodes.getMemberNames()) {

5. distances[node] = numeric\_limits<double>::infinity();

6. }

7. // 起点的距离设为 0

8. distances[startId] = 0;

9. // 将起点加入优先队列

10. pq.push({0, startId});

* + - * distances：存储每个节点的当前最短路径。
      * previous：用于记录每个节点的前驱节点，以便在找到终点时反向构建路径。
      * pq.push({0, startId})：将起点（距离为 0）加入到优先队列中。
    1. 从优先队列中获取最短路径节点

进入主循环，每次从优先队列中弹出当前最短路径的节点（堆顶节点），然后检查它的邻居节点并进行松弛操作（即更新邻居节点的最短路径）。

1. while (!pq.empty()) {

2. double currentDist = pq.top().first; // 当前节点的距离

3. string currentNode = pq.top().second; // 当前节点的 ID

4. pq.pop(); // 从堆中弹出当前节点

5. if (currentNode == endId) break; // 如果已经找到终点，退出循环

6. // 遍历当前节点的邻居节点

7. for (const auto& neighbor : adjacencyMap[currentNode]) {

8. string neighborId = neighbor.first; // 邻居节点的 ID

9. double edgeWeight = neighbor.second; // 边的权重（即距离）

10. double newDist = currentDist + edgeWeight; // 计算到邻居节点的新距离

11.

12. if (newDist < distances[neighborId]) { // 如果发现更短的路径

13. distances[neighborId] = newDist; // 更新距离

14. previous[neighborId] = currentNode; // 更新前驱节点

15. pq.push({newDist, neighborId}); // 将邻居节点重新加入优先队列

16. }

17. }

18. }

* + - * **松弛操作**：对于当前节点的每个邻居，计算通过当前节点到达邻居的距离 newDist，如果 newDist 小于原有的距离，就更新该邻居的最短距离，并将邻居节点重新加入优先队列。

**路径计算参数**：路径的权重是基于相邻节点之间的距离，本次项目中距离的计算使用 **哈弗辛公式**，这是一个考虑地球曲率的地理距离计算方法。

Haversine公式的具体形式为：

其中：

* Δϕ 是两点纬度差（弧度），
* Δλ是两点经度差（弧度），
* R 是地球半径（单位为米），
* d是两点之间的距离。

具体实现如下：

1. double calculateDistance(double lat1, double lon1, double lat2, double lon2) {

2. const double R = 6371e3;

3. double phi1 = lat1 \* M\_PI / 180;

4. double phi2 = lat2 \* M\_PI / 180;

5. double deltaPhi = (lat2 - lat1) \* M\_PI / 180;

6. double deltaLambda = (lon2 - lon1) \* M\_PI / 180;

7.

8. double a = sin(deltaPhi / 2) \* sin(deltaPhi / 2) +

9. cos(phi1) \* cos(phi2) \* sin(deltaLambda / 2) \* sin(deltaLambda / 2);

10. double c = 2 \* atan2(sqrt(a), sqrt(1 - a));

11.

12. return R \* c;

13. }

14.

**返回结果**

一旦算法找到终点节点（或者堆为空时没有找到路径），我们通过 previous 数组反向构建从起点到终点的最短路径，并将路径以 JSON 格式传递给前端显示。

1. Json::Value path;

2. if (distances[endId] == numeric\_limits<double>::infinity()) {

3. return path; // 如果没有路径，返回空路径

4. }

5.

6. // 从终点反向构建路径

7. string currentNode = endId;

8. while (currentNode != startId) {

9. if (previous.find(currentNode) == previous.end()) {

10. path.clear(); // 如果没有有效路径，清空并返回

11. return path;

12. }

13. Json::Value pathNode;

14. pathNode["id"] = currentNode;

15. pathNode["lat"] = nodes[currentNode]["lat"];

16. pathNode["lon"] = nodes[currentNode]["lon"];

17. path.append(pathNode);

18. currentNode = previous[currentNode];

19. }

20.

21. // 将起点加入路径

22. Json::Value startNode;

23. startNode["id"] = startId;

24. startNode["lat"] = nodes[startId]["lat"];

25. startNode["lon"] = nodes[startId]["lon"];

26. path.append(startNode);

27.

28. // 反转路径，因为我们是从终点开始构建的

29. std::reverse(path.begin(), path.end());

30. return path;

31.

### 前后端交互

​ 在本项目中，前端和后端通过 JavaScript 和 WebAssembly (Wasm) 实现交互，前端通过 Emscripten 提供的绑定接口调用后端 C++ 实现的功能，后端则根据请求进行处理，并将结果返回给前端。这部分主体是在示例代码的基础上进行的修改。

​ 数据格式主要采用 JSON，后端使用 Json::Value 存储数据，并通过 Json::StreamWriterBuilder 将数据转化为字符串发送给前端。前端通过 JSON.parse() 解析字符串，并在页面中显示。

* **数据请求与加载**
  + 前端启动时，通过调用 Module.load() 函数请求加载后端数据。load() 函数会从 XML 文件（中解析节点（node）和道路（way）信息，存储在后端的 JSON 数据结构中。
  + 加载完成后，前端通过调用 Module.getNodes(), Module.getWays() 和 Module.getBuildings() 函数来获取地图中的节点、道路和建筑物信息，并用这些数据进行图形的绘制。
* **数据结构与交互**

前端和后端之间使用 JSON 格式 作为数据交换格式，后端将解析后的数据通过 Json::Value 类型存储，在调用接口时，前端通过 Emscripten 的绑定机制，将这些数据转化为 JSON 字符串 发送给前端，前端再解析这些 JSON 字符串并进行相应的展示和操作。

* **后端获取节点数据：**

1. string getNodes() {

2. Json::StreamWriterBuilder builder;

3. return Json::writeString(builder, nodes);

4. }

5.

* + **前端处理节点数据：**

1. var nodesJson = Module.getNodes();

2. var nodesData = JSON.parse(nodesJson);

3.

解析返回的 JSON 数据，并将节点信息绘制在地图上。

* **路径规划与计算**
  + 前端调用后端提供的 getShortestPath() 函数，传入起点、终点以及行驶模式（步行或驾车）等参数。

1. var pathJson = Module.getShortestPath(startId, endId, isPedestrianMode);

2. var pathData = JSON.parse(pathJson);

3. 后端使用 Dijkstra 算法 根据图的邻接表计算最短路径，并将结果（路径）返回给前端。

4. string getShortestPath(const string& startId, const string& endId, bool isPedestrian) {

5. Json::StreamWriterBuilder builder;

6. Json::Value path = findShortestPath(startId, endId, isPedestrian);

7. return Json::writeString(builder, path);

8. }

9.

### 前端界面

本次项目的前端界面设计实现了地图浏览、路径规划、模式选择等功能。主要包括以下几个模块：

* **地图显示模块**：用于展示城市地图，并在上面标出起点、终点及路径。

地图区域是界面的核心部分，占据整个页面的主要空间。地图区域支持：

* + **交互式缩放和拖动**：用户可以通过鼠标滚轮、点击和拖动地图来查看不同区域。
  + **节点标记**：在地图上，用户可以自由选择的起点和终点。若用户选择的点不在所提供的点上，则系统会自动将该点并到最近的一个节点上，然后规划路径。
  + **路径展示**：计算出的最短路径将通过绿色路线在地图上展示。
  + **模式切换**：允许用户在步行模式和驾车模式之间进行切换。在界面顶部放置一个明显的按钮，允许用户选择步行模式或驾车模式。按钮切换后，界面会自动更新路径规划的规则。
* **信息展示模块**：显示路径信息、计算结果等。

信息展示模块位于页面的右下部，用于显示当前路径的相关信息。

* + **起点与终点的详细信息**：坐标、序号等。
  + **路径信息**：包括总路径规划花费的时间。
  + **错误提示**：如果无法找到路径，系统会弹出信息框提示用户“没有可达路径”。

### 性能考虑

1. **时间复杂度分析**

本次最短路径规划采取的关键算法为Dijkstra 算法的堆优化版本。

Dijkstra 算法的时间复杂度主要由两部分组成：

* + 邻接表遍历：对于每个节点，其所有相邻节点都需要被检查，因此在最坏情况下，遍历的时间复杂度为 O(E)，其中 E 是图中边的数量。
  + 优先队列操作：Dijkstra 算法使用优先队列来选择最短路径节点，优先队列的插入和删除操作时间复杂度为 O(log V)，其中 V 是节点数。因此，Dijkstra 算法的总体时间复杂度为 O((V + E) \* log V)，适用于稀疏图。

1. **空间复杂度分析**
   * 本次存储节点和边主要采用的方法为邻接表存储（map<string, vector<pair<string, double>>>）。

邻接表的存储复杂度为 O(V + E)，其中 V 是节点数，E 是边数。相比于邻接表的 O(V²) 存储，邻接表在处理稀疏图时具有更高的存储效率。

* + 最短路径辅助存储使用了：
    - distances 数组：存储每个节点到起点的最短距离。需要 O(V) 空间。
    - previous 数组：存储每个节点的前驱节点，用于重建最短路径。需要 O(V) 空间。
  + Dijkstra 算法使用一个优先队列来处理当前未访问节点，并按最短距离排序。队列中的元素个数最多为 V，因此需要 O(V) 空间。

总体的空间复杂度为 O(V + E)。

**性能瓶颈与优化**

​ 由于本次项目老师在课上要求使用的规划方法为Dijkstra算法的堆优化版本，故本次项目的主体即采用此算法，但是实际上还可以进一步进行优化。同时，受限于我自己电脑的性能以及代码的能力，很多地方的性能都并没有达到充分的快，故在此部分我将对本次项目中的性能瓶颈进行分析，并且尝试对其优化。

​ 随着节点和边数的增加，图的规模将直接影响算法的执行时间和空间需求。对于非常大的图，Dijkstra 算法的性能会受到影响。但是，由于OSM网站对于下载量的限制以及我自己电脑CPU性能的限制（在尝试下载更大的地图数据后运行程序对于内存不足的报错），采用Dijkstra来应对我本次项目中较小的数据量仍然是可行的。然而，在现实情况下，设计足以应对大规模数据的算法仍然是十分必要的，故以下我将讨论几种设想的解决办法以及具体实现。

#### 采用 ****Fibonacci 堆****

​ 通过采用 **Fibonacci 堆**（FibonacciHeap）替代堆实现的优先队列（priority\_queue），可以在 Dijkstra 算法中更加高效地选择下一个最短路径节点。

​ 斐波那契堆是基于斐波那契数列（Fibonacci sequence）的一种堆结构，它由一个由树组成的集合（树的根节点）和一些支持的操作组成。与其他堆不同的是，斐波那契堆中的树是无序的，而且树的结构是通过懒惰合并策略进行延迟调整的，从而降低了时间复杂度。

斐波那契堆的特点：

* **松散结构**：斐波那契堆由一组根节点树组成，并且每棵树的结构并不严格。树的数量可以很多，树的形状不完全平衡，因此操作上可以灵活处理。
* **延迟合并**：与其他堆相比，斐波那契堆更注重“懒”更新，很多操作并不是立刻进行，而是推迟到必须进行的时候才去做。这种延迟策略使得某些操作的平均时间复杂度降低。

本次所需要的斐波那契堆的操作：

1. **插入（Insert）**：插入一个新的元素时，首先创建一个包含该元素的新树并将其添加到根节点集合中。这个操作的时间复杂度是 **O(1)**。
2. **最小元素查找（Find-Min）**：找到堆中最小的元素（即最小根节点）。由于堆是由一组树组成的，每次查询时直接检查根节点即可。时间复杂度是 **O(1)**。
3. **删除最小元素（Delete-Min）**：删除堆中的最小元素，并将其所有子节点加入堆中。此操作的时间复杂度为 **O(log n)**。这是因为需要重新合并堆中的树，确保堆的性质。

​ 该部分代码我通过参考网上的资料另外实现了，和本次项目主体代码的唯一区分就是Dijkstra算法中的优先队列由自己实现的斐波那契堆替代了。

最后效果示例如下，由于整体数据量较小，故性能提升的差异并不大。

本次优化实现斐波那契堆的关键代码如下：

1. class FibNode

2. {

3. public:

4. int key; // 关键字(键值)

5. int degree; // 度数

6. std::string name; // 节点名称

7. FibNode\* left; // 左兄弟

8. FibNode\* right; // 右兄弟

9. FibNode\* child; // 第一个孩子节点

10. FibNode\* parent; // 父节点

11. bool marked; // 是否被删除第一个孩子

12.

13. FibNode(int key, std::string n)

14. {

15. this->key = key;

16. this->degree = 0;

17. this->name = n;

18. this->marked = false;

19. this->left = this;

20. this->right = this;

21. this->parent = nullptr;

22. this->child = nullptr;

23. }

24. };

25.

26. class FibHeap {

27. private:

28. int keyNum; // Total number of nodes in the heap

29. FibNode\* min; // Pointer to the minimum node (root of the heap)

30.

31. public:

32. // Constructor initializing keyNum and min

33. FibHeap() : keyNum(0), min(nullptr) {}

34.

35. public: bool empty()

36. {

37. return !this->keyNum;

38. }

39.

40. // Method to remove a node from a doubly linked list

41. private: void removeNode(FibNode\* node) {

42. node->left->right = node->right;

43. node->right->left = node->left;

44. }

45.

46. // Method to add a node to the root list

47. private: void addNode(FibNode\* node, FibNode\* root) {

48. node->left = root->left;

49. root->left->right = node;

50. node->right = root;

51. root->left = node;

52. }

53.

54. // Method to insert a node into the Fibonacci heap

55. private: void insert(FibNode\* node) {

56. if (keyNum == 0)

57. min = node;

58. else {

59. addNode(node, min);

60. if (node->key < min->key)

61. min = node;

62. }

63. keyNum++;

64. }

65.

66. // Method to insert a node with a specific key and name

67. public: void insert(int key, std::string n) {

68. FibNode\* node = new FibNode(key, n);

69. if (node == nullptr) return;

70. insert(node);

71. }

72.

73. // Method to concatenate two doubly linked lists

74. private: void catList(FibNode\* a, FibNode\* b) {

75. FibNode\* tmp = a->right;

76. a->right = b->right;

77. b->right->left = a;

78. b->right = tmp;

79. tmp->left = b;

80. }

81.

82. // Method to extract the minimum node from the heap

83. private: FibNode\* extractMin() {

84. FibNode\* p = min;

85. if (p == p->right)

86. min = nullptr;

87. else {

88. removeNode(p);

89. min = p->right;

90. }

91. p->left = p->right = p;

92. return p;

93. }

94.

95. // Method to link two nodes in the heap

96. private: void link(FibNode\* node, FibNode\* root) {

97. removeNode(node);

98. if (root->child == nullptr)

99. root->child = node;

100. else

101. addNode(node, root->child);

102.

103. node->parent = root;

104. root->degree++;

105. node->marked = false;

106. }

107.

108. // Method to consolidate the heap and merge trees of the same degree

109. private: void consolidate() {

110. int maxDegree = (int)std::floor(std::log(keyNum) / std::log(2.0));

111. int D = maxDegree + 1;

112. std::vector<FibNode\*> cons(D + 1, nullptr); // Vector initialized with nullptr

113.

114. while (min != nullptr) {

115. FibNode\* x = extractMin();

116. int d = x->degree;

117. while (cons[d] != nullptr) {

118. FibNode\* y = cons[d];

119. if (x->key > y->key) {

120. std::swap(x, y);

121. }

122. link(y, x);

123. cons[d] = nullptr;

124. d++;

125. }

126. cons[d] = x;

127. }

128.

129. min = nullptr;

130. for (int i = 0; i < D; i++) {

131. if (cons[i] != nullptr) {

132. if (min == nullptr)

133. min = cons[i];

134. else {

135. addNode(cons[i], min);

136. if (cons[i]->key < min->key)

137. min = cons[i];

138. }

139. }

140. }

141. }

142.

143.

144. // Method to remove the minimum node and return its value

145. public: std::pair<double, std::string> removeMin() {

146. if (min == nullptr)

147. return std::pair<double, std::string>(-1, "");

148.

149. FibNode\* m = min;

150. double tmpkey = m->key;

151. std::string tmpn = m->name;

152.

153. while (m->child != nullptr) {

154. FibNode\* child = m->child;

155. removeNode(child);

156. if (child->right == child)

157. m->child = nullptr;

158. else

159. m->child = child->right;

160.

161. addNode(child, min);

162. child->parent = nullptr;

163. }

164.

165. removeNode(m);

166. if (m->right == m)

167. min = nullptr;

168. else {

169. min = m->right;

170. consolidate();

171. }

172. keyNum--;

173.

174. return std::pair<double, std::string>(tmpkey, tmpn);

175. }

176.

177. // Method to get the minimum key in the heap

178. public: int minimum() {

179. if (min == nullptr)

180. return -1;

181. return min->key;

182. }

183.

184. // Method to destroy the heap

185. private: void destroyNode(FibNode\* node) {

186. if (node == nullptr) return;

187.

188. FibNode\* start = node;

189. do {

190. destroyNode(node->child);

191. node = node->right;

192. node->left = nullptr;

193. } while (node != start);

194. }

195.

196. // Destroy the whole heap

197. public: void destroy() {

198. destroyNode(min);

199. }

200. };

#### 采用 ****A\* 算法****

​ 通过采用 **A\* 算法**（基于启发式搜索）替代 Dijkstra 算法，尤其是在大规模图中，通过启发式函数可以减少不必要的路径搜索，可以大大提高最短路径搜索的时间效率。

​ A\*（A-star）算法是一种用于图形搜索和路径规划的启发式搜索算法，它能够高效地找到从起点到终点的最短路径。它结合了 **广度优先搜索**（BFS）和 **贪婪最佳优先搜索**（Greedy Best-First-Search）的优点，通过合理的启发式函数来引导搜索，避免了盲目遍历。 这个算法的核心思想是，每次选择一个“最有前途”的节点进行扩展，即优先选择估计总代价最小的节点。

​ 其**核心概念**有：

* 实际代价（gScore[n]）:
  + 从起点到当前节点 n 的最短已知路径代价。
  + 初始值为无穷大（Infinity），起点的 gScore[start] 设置为 0。
* 估算代价（h(n)）:
  + 启发式函数，估算从当前节点 n 到目标节点的代价。
  + 常见的 h(n) 是欧几里得距离或曼哈顿距离。
* 总代价（fScore[n]）:
  + 表示从起点经过当前节点到终点的总代价估算。A\*算法通过不断扩展最小 **f(n)** 的节点，最终找到从起点到终点的最优路径。 $$ f(n)=gScore[n]+h(n) $$
  + fScore[start] 初始化为 h(start)。
* 开放集（openSet）:
  + 存储所有待处理的节点，优先处理总代价估算最小的节点。
  + 实现时通常用最小堆（min-heap）或优先队列。本次用priority\_queue实现。

其**大致流程**为：

* + 初始化
    - 定义 openSet，只包含起点 start。
    - 定义 cameFrom，记录每个节点的前驱节点，初始化为空。
    - 初始化 gScore，起点的 gScore[start] = 0，其他节点为无穷大。
    - 初始化 fScore，起点的 fScore[start] = h(start)，其他节点为无穷大。
  + 主循环

当 openSet 不为空时，执行以下操作：

a. 从开放集中选择总代价最小的节点

* + - 从 openSet 中取出 fScore 值最小的节点，称为当前节点 current。
    - 如果 current

是目标节点 goal：

* + - * 调用 reconstruct\_path(cameFrom, current) 返回路径。
      * 算法结束。

b. 从开放集中移除当前节点

* + - 当前节点 current 处理完毕，从 openSet 中移除。

c. 遍历当前节点的所有邻居

对于每个邻居 neighbor：

* + - 计算从起点到邻居 neighbor 的临时实际代价：
    - 比较 tentative\_gScore 和当前记录的 gScore[neighbor]：
      * 如果 $$ tentative < gScore[neighbor] $$
        + 更新 gScore[neighbor] 为 tentative\_gScore。
        + 更新 fScore[neighbor] 为： fScore[neighbor]=gScore[neighbor]+h(neighbor)
        + 更新 cameFrom[neighbor]，记录邻居的前驱节点为 current。
    - 如果邻居 neighbor 不在 openSet 中：
      * 将邻居加入 openSet。
  + 搜索失败

如果 openSet 为空且未找到路径：

* + - 返回失败（failure）。
  + 路径重建

当找到目标节点后，通过 cameFrom 记录的前驱节点回溯，重建从起点到终点的路径。

A\*算法的伪代码：

1. function reconstruct\_path(cameFrom, current)

2. total\_path := {current}

3. while current in cameFrom.Keys:

4. current := cameFrom[current]

5. total\_path.prepend(current)

6. return total\_path

8. // A\* finds a path from start to goal.

9. // h is the heuristic function. h(n) estimates the cost to reach goal from node n.

10. function A\_Star(start, goal, h)

11. // The set of discovered nodes that may need to be (re-)expanded.

12. // Initially, only the start node is known.

13. // This is usually implemented as a min-heap or priority queue rather than a hash-set.

14. openSet := {start}

15.

16. // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from the start

17. // to n currently known.

18. cameFrom := an empty map

19.

20. // For node n, gScore[n] is the currently known cost of the cheapest path from start to n.

21. gScore := map with default value of Infinity

22. gScore[start] := 0

23.

24. // For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to

25. // how cheap a path could be from start to finish if it goes through n.

26. fScore := map with default value of Infinity

27. fScore[start] := h(start)

28.

29. while openSet is not empty

30. // This operation can occur in O(Log(N)) time if openSet is a min-heap or a priority queue

31. current := the node in openSet having the lowest fScore[] value

32. if current = goal

33. return reconstruct\_path(cameFrom, current)

34.

35. openSet.Remove(current)

36. for each neighbor of current

37. // d(current,neighbor) is the weight of the edge from current to neighbor

38. // tentative\_gScore is the distance from start to the neighbor through current

39. tentative\_gScore := gScore[current] + d(current, neighbor)

40. if tentative\_gScore < gScore[neighbor]

41. // This path to neighbor is better than any previous one. Record it!

42. cameFrom[neighbor] := current

43. gScore[neighbor] := tentative\_gScore

44. fScore[neighbor] := tentative\_gScore + h(neighbor)

45. if neighbor not in openSet

46. openSet.add(neighbor)

47.

48. // Open set is empty but goal was never reached

49. return failure

**与Dijkstra算法的比较**

* Dijkstra算法计算源点到其他所有点的最短路径长度，A\*关注点到点的最短路径(包括具体路径)。
* Dijkstra算法的实质是广度优先搜索，是一种发散式的搜索，所以空间复杂度和时间复杂度都比较高。对路径上的当前点，A\*算法不但记录其到源点的代价，还计算当前点到目标点的期望代价，是一种启发式算法，也可以认为是一种深度优先的算法。
* 由第一点，当目标点很多时，A \*算法会带入大量重复数据和复杂的估价函数，所以如果对于许多个目标点且不要求获得具体路径而只比较路径长度时，Dijkstra算法会成为更好的选择。但是本次项目中只有一个目标点，故A \*算法的效率会更高。

​ 该部分代码我同样通过参考网上的资料另外实现了，和本次项目主体代码的区分体现于原本的findShortedPath函数被findShortestPathAStar函数所替代，其余部分保持不变，同时A\*算法中的优先队列依旧采用的是STL库中堆实现的优先队列priority\_queue。

最后效果示例如下，相较于原本的堆优化的Dijkstra算法以及采用Fibonacci堆的Dijkstra算法查找的时间效率有了非常显著的提升。

本次实现A\*算法的函数代码如下：

1. Json::Value findShortestPathAStar(const string& startId, const string& endId, bool isPedestrian) {

2. auto& adjacencyMap = isPedestrian ? pedestrianAdjacencyMap : vehicleAdjacencyMap;

3.

4. // 使用 unordered\_set 替代 set

5. unordered\_set<string> closedSet;

6.

7. priority\_queue<pair<double, string>, vector<pair<double, string>>, greater<>> openSet;

8.

9. map<string, double> gScore; // 从起点到当前节点的实际代价

10. map<string, double> fScore; // 估算总代价

11. map<string, string> previous;

12.

13. // 初始化起点

14. gScore[startId] = 0;

15. fScore[startId] = calculateDistance(

16. nodes[startId]["lat"].asDouble(),

17. nodes[startId]["lon"].asDouble(),

18. nodes[endId]["lat"].asDouble(),

19. nodes[endId]["lon"].asDouble()

20. );

21. openSet.push({fScore[startId], startId});

22.

23. while (!openSet.empty()) {

24. string currentNode = openSet.top().second;

25. openSet.pop();

26.

27. if (currentNode == endId) {

28. // 重建路径

29. Json::Value path;

30. while (currentNode != startId) {

31. Json::Value pathNode;

32. pathNode["id"] = currentNode;

33. pathNode["lat"] = nodes[currentNode]["lat"];

34. pathNode["lon"] = nodes[currentNode]["lon"];

35. path.append(pathNode);

36. currentNode = previous[currentNode];

37. }

38.

39. Json::Value startNode;

40. startNode["id"] = startId;

41. startNode["lat"] = nodes[startId]["lat"];

42. startNode["lon"] = nodes[startId]["lon"];

43. path.append(startNode);

44.

45. std::reverse(path.begin(), path.end());

46. return path;

47. }

48.

49. // 注意：使用 unordered\_set 时，判断是否存在的方法略有不同

50. closedSet.insert(currentNode);

51.

52. for (const auto& neighbor : adjacencyMap[currentNode]) {

53. string neighborId = neighbor.first;

54. double edgeWeight = neighbor.second;

55.

56. // 使用 count() 判断是否在 closedSet 中

57. if (closedSet.count(neighborId) > 0) continue;

58.

59. double tentativeGScore = gScore[currentNode] + edgeWeight;

60.

61. if (gScore.count(neighborId) == 0 || tentativeGScore < gScore[neighborId]) {

62. // 找到更好的路径

63. previous[neighborId] = currentNode;

64. gScore[neighborId] = tentativeGScore;

65.

66. // 估算到终点的启发式距离

67. double heuristic = calculateDistance(

68. nodes[neighborId]["lat"].asDouble(),

69. nodes[neighborId]["lon"].asDouble(),

70. nodes[endId]["lat"].asDouble(),

71. nodes[endId]["lon"].asDouble()

72. );

73.

74. fScore[neighborId] = gScore[neighborId] + heuristic;

75. openSet.push({fScore[neighborId], neighborId});

76. }

77. }

78. }

79.

80. // 没有找到路径

81. return Json::Value();

82. }

#### 多线程和并行计算

​ 本次项目当我尝试扩大地图数据时收到了对于cpu容量不足的报错，由于我代码水平有限，故此处只进行理论上的分析，不给出实际代码实现。

​ 为了提升路径规划的效率，特别是在处理较大地图数据集时，可以引入了多线程和并行计算的优化策略。

1. 在数据加载阶段，使用线程池对节点和道路数据的解析进行分块处理

大致思路：

* + 将地图的 XML 数据按区域或文件大小分块，每个线程负责解析一个块。
  + 采用线程池（std::thread 和 std::async）管理多线程任务，避免线程频繁创建和销毁带来的开销。

1. void loadMapData(const std::string& mapFile) {

2. // 读取 XML 数据并分块

3. auto blocks = divideIntoBlocks(mapFile);

4. // 创建线程池并分配任务

5. std::vector<std::future<void>> tasks;

6. for (auto& block : blocks) {

7. tasks.emplace\_back(std::async(std::launch::async, [this, &block]() {

8. parseBlock(block);

9. }));

10. }

11. // 等待所有线程完成

12. for (auto& task : tasks) {

13. task.get();

14. }

15. }

16.

1. 两种邻接图（pedestrianAdjacencyMap 和 vehicleAdjacencyMap）采用多线程分别处理

例如可以将两种邻接表的构建函数封装为一个，然后采用多线程

1. void constructAdjacencyMaps() {

2. std::thread pedestrianThread(&MapData::constructPedestrianAdjacencyMap, this);

3. std::thread vehicleThread(&MapData::constructVehicleAdjacencyMap, this);

4. pedestrianThread.join();

5. vehicleThread.join();

6. }

7. void constructPedestrianAdjacencyMap() {

8. // 遍历节点和道路，根据行人模式规则构建邻接表

9. }

10. void constructVehicleAdjacencyMap() {

11. // 遍历节点和道路，根据车辆模式规则构建邻接表

12. }

1. 对于起点的邻居节点，分配多个线程分别计算从这些邻居到目标节点的最短路径，最终合并结果

在路径计算中，使用了堆优化的 Dijkstra 算法。如果节点数量巨大，可以利用多源路径并行计算的思想，对于起点的邻居节点，分配多个线程分别计算从这些邻居到目标节点的最短路径，最终合并结果。

1. std::vector<std::future<void>> tasks;

2. void parallelDijkstra(const std::string& startId, const std::string& endId) {

3. auto neighbors = adjacencyMap[startId];

4. for (const auto& neighbor : neighbors) {

5. tasks.emplace\_back(std::async(std::launch::async, [this, &neighbor, endId]() {

6. calculatePartialPath(neighbor.first, endId);

7. }));

8. }

9. for (auto& task : tasks) {

10. task.get();

11. }

12. }

13. void calculatePartialPath(const std::string& startId, const std::string& endId) {

14. // 使用 Dijkstra 算法计算从 startId 到 endId 的路径

15. // 将结果存入线程安全的全局数据结构中

16. }

### 最终扩展

​ 在初步完成地图路径规划项目以后，课程设置了pj展示。在观看了其他同学的pj演示以后，我发现自己当前所实现的地图还是太简陋了，只有点和线，没有详细的地点以及道路表示。于是在询问了助教后，我又在原本所完成的基础上通过调用百度地图的api进一步丰富了我的前端实现。同时，由于百度地图的前端已经提供了整个中国的数据，所以升级后的数据量也扩大了。

​ 由于修改升级前端代码时已经处在期末季，故完成的较为简陋，完整代码见关键代码部分，以下仅列举关键部分。

**<head>部分**：

* 引入了百度地图的SDK脚本。通过<script>标签加载百度地图API。

此处的API key需要自己在百度的开发者平台上申请。同时，要注意应用类别需要选择浏览器端，因为最后是要在网页上打开的。一开始我选择的时服务端，无法使用API。申请好以后界面如下：



* 设置了页面的基本样式，包含了页面布局、控件的样式、地图的容器等。

**<body>部分**：

* **地图容器**（#map-container）：地图的显示区域。我创建了一个百度地图实例，并设置地图的中心为复旦大学位置（经纬度：121.503584, 31.296426）并设置了缩放级别为15。这里我一开始是直接使用的度数为单位的经纬度，发现展示的中心并不是复旦，于是重新在网上查询了复旦经纬度的小数形式。
* **控件区**（.controls）：包括了两个按钮来切换模式（行走和驾驶模式）。这两个按钮和之前实现的是一样的。
* **信息显示区**（.info-display）：用于显示所选坐标点、路径计算时间和图例（地图上不同颜色的标记含义）。这部分也和之前实现的是一样的。
* **路径计算**（calculateRoute）：根据当前的模式（步行或驾驶）来计算从起点到终点的路径。路径计算的时间会被记录并显示在页面上。

## 实验结果

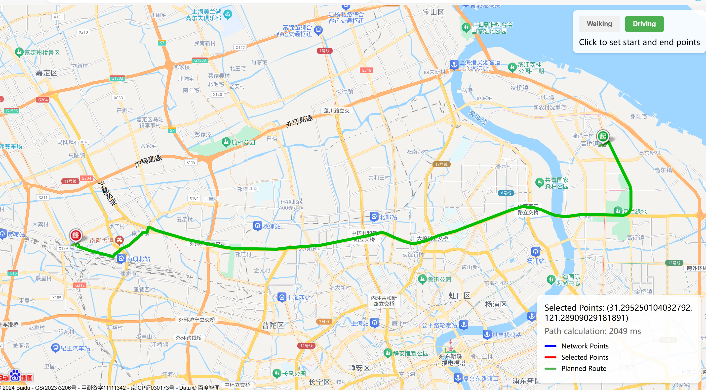
如下图：

* 初始界面
* 随机点击一个点
* 再随机点击一个点，得到最短路径

最终前端升级后：

由于数据量明显增大，我这里路径计算的时间明显从几十毫秒增加到了几百毫秒。尤其是但我把地图的缩放级别调低时，两个距离远的路径规划的时间甚至达到了几千毫秒。

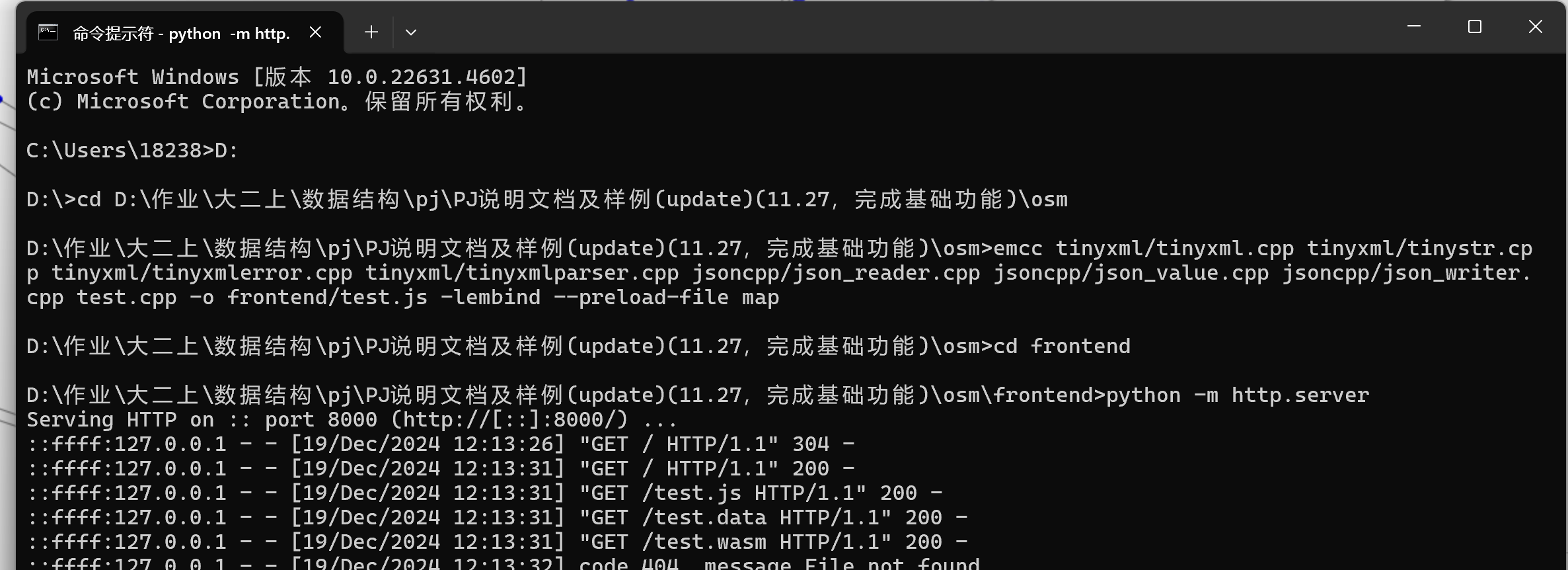




## 项目运行方法

整个项目的运行方法和项目初始的readme文件中所给的方式保持一致。

打开命令提示符cmd如下操作即可：



若发现当前网页打开的是之前的项目内容，Ctrl + shift + R即可。

## 实验遇到的问题与解决方法

1. 在运行了示例代码以后得到预期画面，但是换成自己在OSM上下载的数据后就完全不能跑了，只有一片空白。

**解决办法：**在询问了助教以后知道了需要调整一下html文件中scale函数的具体参数。其中，这个参数的调整和一开始在OSM上选取的四个坐标点有很大关系。再反复尝试几次以后成功得到了想要的画面。

1. 在完成了初步的代码撰写以后想要查看运行的效率，但是不知道怎么把后端代码的速度传到前端。

**解决办法：**在询问了助教以后知道了前端也有可以直接调用的函数用来计时，于是直接修改前端代码。

1. 初步代码中对于way不加以区分，但是实际上OSM的数据中way既可以用来表示建筑，也可以用来表示道路，不加以区分会导致道路规划错误。

**解决办法：**在询问了助教以后知道way的这两个种类并不需要考虑tag，可以直接以way是否闭合这一点进行区分。于是在后端加入buildings这一概念，向前端传输的除了节点、道路还有建筑。

1. 在初步完成地图路径规划项目以后，课程设置了pj展示。在观看了其他同学的pj演示以后，我发现自己当前所实现的地图还是太简陋了，页面展示的并不是地图，更像是草图。

**解决办法：**在询问了助教后，我了解到还可以直接调用现有的api来进一步丰富了前端实现。于是，在查阅了百度地图官方给出了文档以及查阅资料以后，我进一步给我的前端进行了升级。

## 实验心得

​ 通过本次实验，我对路径规划的理论算法和实际开发流程有了更为全面的理解。这不仅让我提前学习了最短路径算法（如Dijkstra算法）及其优化（堆优化、斐波那契堆），还让我首次尝试将这些算法应用于实际问题，为我打开了从理论到实践的新视野。

​ 本次实验从数据获取、路径计算到前后端交互，各个模块划分明确且紧密协作。在数据获取方面，我通过获取OpenStreetMap（OSM）的真实地图数据，学会了如何提取有效的节点和道路信息并构建邻接表。尽管起初对本次实验复杂感到畏惧，但示例代码提供了良好的基础框架，帮助我快速入门。在示例代码的基础上，通过反复阅读示例代码、查阅相关资料并借助GPT等工具的帮助，我成功地将一个静态的长方形转变为一个可交互的路径规划地图。这一过程中，我深刻感受到了理论转化为实践的挑战，同时也为最终实现目标收获了满满的成就感。

​ 在路径规划功能初步实现后，我进一步尝试了两种优化方法：使用斐波那契堆提升Dijkstra算法的效率，以及引入启发式A\* 算法。虽然在小规模数据集上性能差异有限，但这些优化思路为处理大规模地图数据的需求奠定了重要基础。同时，这两种优化方法尤其是A\*算法对时间效率的惊人提升让我更直观地体会到选择合适数据结构和算法的重要性。

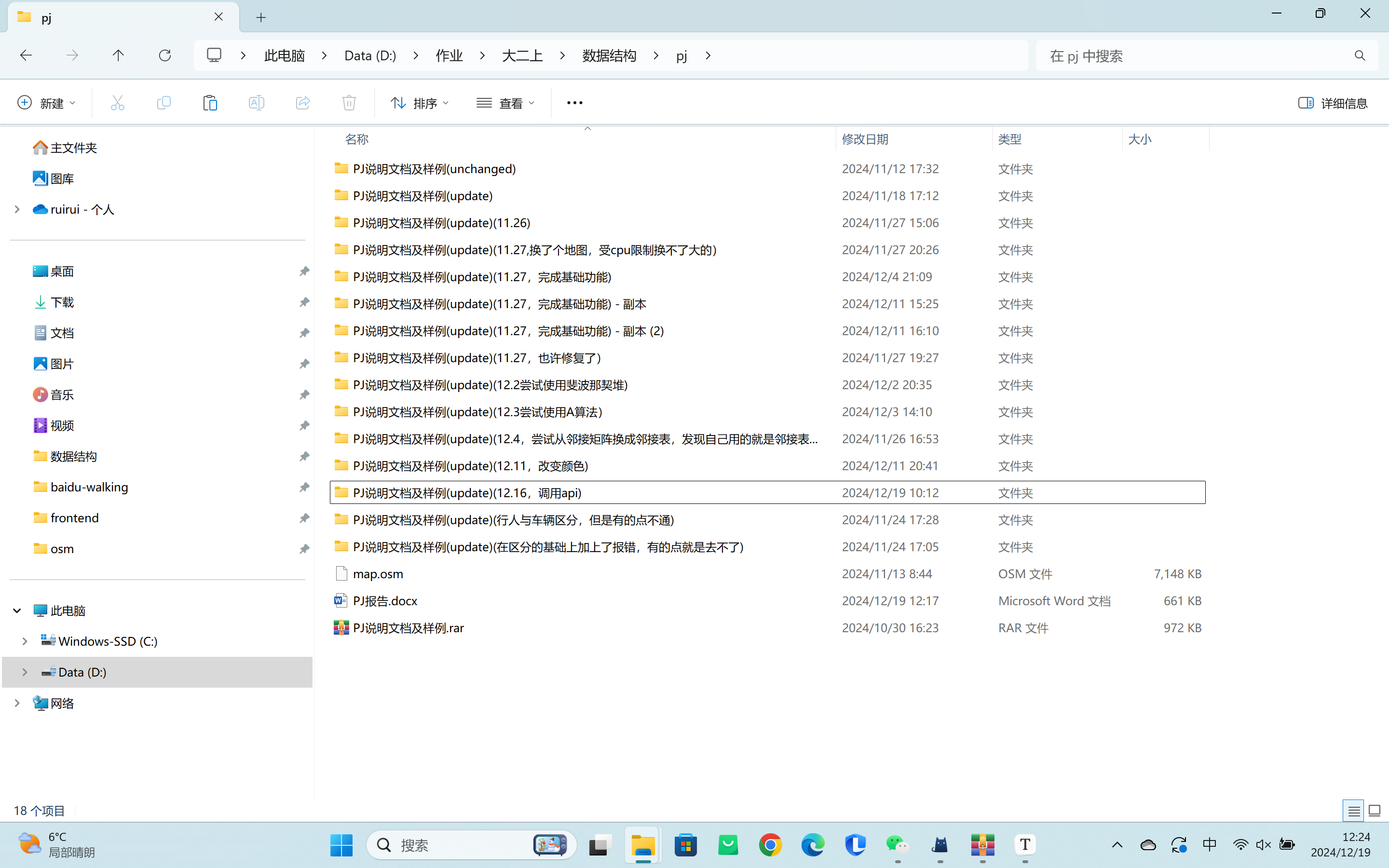
​ 实验过程中，我也遇到了许多挑战。例如由于对JavaScript不够熟悉导致初期在实现前后端交互时感到无从下手、地图数据格式与预期不符导致需要仔细分析数据并调整处理逻辑以及实现代码过程中的各种神奇BUG，等等。这些问题的逐一克服，不仅加深了我对整个项目结构和逻辑的理解，也显著提升了我的调试能力和面对问题的耐心。

​ 在完成实验的过程中，我还尝试在基本要求的基础上加入拓展功能。例如，我尝试在地图交互功能的基础上加入模式切换功能，为用户提供更加直观的操作体验；同时将系统的功能从只能选择给定的节点拓展到了可以选择地图上的任意节点，让用户的选择更加丰富。

​ 在实验的最后，通过调用百度地图的API，我进一步丰富了自己的前端界面。通过结合百度地图的各类功能，我为用户提供了详细精细的地图页面、不同颜色的标记解释，使得页面更加直观、易懂。

​ 总的来说，这次实验让我深刻认识到数据结构与算法在实际开发中的作用，并通过解决实际问题积累了宝贵的经验，增强了我面对未来开发挑战的信心。我期待能够将这些经验应用到更大规模、更复杂的项目中，不断提升自己的能力。

实验过程中的不断尝试：



## 参考资料

本次编写代码的过程中遇到了很多困难，最后好多都是询问助教以后得到的的解决方向，感谢助教！

本次代码中所有涉及前端以及前后端交互的部分都是由[ChatGPT](https://chatgpt.com/)和[Claude](https://claude.ai/)这两个大语言模型所撰写的，感谢它们！

本次对前端升级的部分是通过调用百度地图的API实现的，感谢百度地图愿意提供免费额度！

本次性能优化中Fibonacci堆部分代码参考了cnblogs网站上luanxm的文章[数据结构——斐波那契堆](https://www.cnblogs.com/luanxm/p/10848032.html)中对斐波那契堆的讲解以及部分功能的实现。

本次性能优化中A\* 算法部分代码参考了[维基百科]([搜索算法-维基百科，自由的百科全书 --- A\* search algorithm - Wikipedia](https://en.wikipedia.org/wiki/A*_search_algorithm))上对A\*算法的讲解以及对应的伪代码。

本次代码中最终拓展升级部分的前端代码参考了CSDN网站上[王同学要努力](https://wcaicai.blog.csdn.net/)的文章[百度地图API的使用教程以及案例](https://blog.csdn.net/m0_46374969/article/details/118295186)。