基于图论的自动驾驶全局路径规划

陈潇杰,202100120059

June 12, 2024

Abstract

自动驾驶中的决策规划模块是衡量和评价自动驾驶能力最核心的指标之一，它的主要任务是在接收到传感器的各种感知信息之后，对当前环境作出分析，然后对底层控制模块下达指令。典型的决策规划模块可以分为三个层次：全局路径规划、行为决策、运动规划。全局路径规划是指在给定车辆当前位置与终点目标后，通过搜索选择一条最优的路径。本文将采用图论中的 Dijkstra 算法和 A*（A star）算法对自动驾驶中的全局路径规划问题进行建模与求解。相关代码在附录中给出，或者在

1 引言

自动驾驶技术的发展已经使得无人驾驶汽车成为了现实，其关键之一在于决策规划模块的设计与优化。该模块负责从传感器获取的信息中分析环境，为底层控制模块提供指令。在这个模块中，全局路径规划起着关键的作用，它的任务是在给定起点和终点的情况下，选择一条最佳路径。典型的决策规划模块通常包括全局路径规划、行为决策和运动规划三个层次。本文将聚焦于全局路径规划，采用图论中的经典算法，如 Dijkstra 算法 [1] 和 A* 算法 [2]，对自动驾驶中的全局路径规划问题进行建模与求解。通过这项研究，我们旨在深入探讨全局路径规划在自动驾驶系统中的关键作用，以及如何利用经典算法来解决这一复杂而关键的问题。

本文的组织如下，在第二章我们将介绍所使用的两种路径规划算法，Dijkstra 算法和 A* 算法。第三章，我们将对问题进行描述和建模，两种算法的求解将被给出。最后的第四章，我们对文章进行了总结。相关代码被附在附录和 Github: https://github.com/w1ndchan/Route-Planning 中。

2 算法介绍

2.1 Dijkstra 算法

Dijkstra 算法是一种经典的图算法，用于解决单源最短路径问题，即在一个加权有向图或无向图中，从给定的起始顶点到所有其他顶点的最短路径问题。该算法由荷兰计算机科学家艾兹赫尔 · 戴克斯特拉（Edsger W. Dijkstra）于 1956 年提出。以下是 Dijkstra 算法的详细步骤：初始化：首先，将起始顶点的距离设为 0，将所有其他顶点的距离设为无穷大。将起始顶点标记为 “当前顶点” 。对于当前顶点的所有邻接顶点，计算从起始顶点经过当前顶点到达这些邻接顶点的距离。如果经当前顶点到达某个邻接顶点的路径距离比当前记录的距离要短，则更新该邻接顶点的距离为新的最短路径距离。从所有未标记的顶点中选择一个距离起始顶点最短的顶点作为下一个 “当前顶点”，将其标记为已访问。重复上述步骤，直到所有顶点都被标记为已访问，或者目标顶

Algorithm 1 Dijkstra algorithm

Input: Weighted directed or undirected graphs G and start poing T.Assume label of T is 0;

Output: Minimum distance from the starting point to each point $d[n]$ and optimal path $r[n]$;

Initialize the distance, set $d[0] = 0, d[n] = Inf(n \neq 0), r[n] = -1, visited[n] = false$;
Push $< d[0], 0 >$ into the priority queue pq;
while pq is not empty **do**
  $dis \leftarrow pq.top().first(), u \leftarrow pq.top().second()$;
  $pq.pop()$;
  if $dis > d[u]$ **then**
    $continue$;
  end if
  Set $visited[u]$ to $true$;
  for each $v \in [0, n)$ **do**
    if $visited[v]$ is false and $G[u][v] \neq Inf$ **then**
      if $d[u] + G[u][v] < d[v]$ **then**
        $d[v] \leftarrow d[u] + G[u][v]$
        $r[v] \leftarrow u$
        push $< d[v], v >$ into the pq
      end if
    end if
  end for
end while
return $d[n], r[n]$

点的距离被确定为最短路径。当所有顶点的最短路径距离确定后，算法结束。最短路径可以通过回溯每个顶点的前驱顶点来重建。算法的详细步骤在 Algorithm1 中给出。

Dijkstra 算法的时间复杂度取决于所使用的数据结构，传统的 Dijkstra 算法的时间复杂度为 $O(n^2)$。若使用堆优化可以改进时间复杂度。在一般情况下，如果使用最小堆（binary heap）或斐波那契堆（Fibonacci heap）等高效的优先队列实现，则 Dijkstra 算法的时间复杂度为 $O((V + E)logV)$，其中 V 是顶点数，E 是边数。

2.2 A* 算法

Dijkstra 算法具有完备性，即它可以从连通图中找出最优的路径。然而，实际应用场景中，图将更为复杂，节点个数庞大且连接稠密。在这样的场景下使用 Dijkstra 算法的效率是比较低的，因此需要寻找一种效率更高的搜索算法。为了解决 Djikstra 算法的搜索效率问题，Stanford 研究院的 Peter Hart, Nils Nilsson 以及 Bertram Raphael 提出了 A* 算法。A* 算法是一种在图形搜索和路径规划中常用的启发式搜索算法。它可以在给定的图形中找到从起点到终点的最佳路径，通过在搜索过程中使用启发式信息来优化搜索方向。A* 算法结合了 Dijkstra 算法的完备性和贪婪最佳优先搜索的高效性，因此在实践中被广泛应用。

A* 算法通常应用于状态空间图，其中节点表示搜索空间中的状态，边表示状态之间的转换。A 算法需要一个启发函数，用来估计从当前状态到目标状态的最优路径的代价。这个启发函数是

Algorithm 2 A* Algorithm

Input: Weighted directed or undirected graphs, the start&end node;

Output: the optimal path

$openSet \leftarrow \{start\}$
$closedSet \leftarrow \{\}$
$start.g \leftarrow 0$
$start.f \leftarrow start.g + heuristic(start, goal)$
while *openSet* is not empty **do**
  $current \leftarrow$ node in *openSet* with the lowest f value
  if $current = goal$ **then**
    return reconstructPath(*current*)
  end if
  remove *current* from *openSet*
  add *current* to *closedSet*
  for all *neighbor* of *current* **do**
    if *neighbor* is in *closedSet* **then**
      continue
    end if
    $tentative\_gScore \leftarrow current.g + distance(current, neighbor)$
    if *neighbor* is not in *openSet* **then**
      add *neighbor* to *openSet*
    else if $tentative\_gScore \geq neighbor.g$ **then**
      continue
    end if
    $neighbor.parent \leftarrow current$
    $neighbor.g \leftarrow tentative\_gScore$
    $neighbor.f \leftarrow neighbor.g + heuristic(neighbor, goal)$
  end for
end while
return "No path found"

A* 算法中的关键，它决定了搜索过程中每一步的方向。A* 算法使用评估函数来决定下一个要扩展的节点。评估函数通常是启发函数和从起点到当前节点的实际代价的和。A* 算法使用一个路径优劣评价公式为：

$$f(n) = g(n) + h(n), \tag{1}$$

其中，$f(n)$ 是从初始状态经由状态 n 到目标状态的代价估计，$g(n)$ 是从初始状态到状态 n 的实际代价，$h(n)$ 是从状态 n 到目标状态的最佳路径的估计代价。A* 算法需要维护两个状态表，分别称为 openList 表和 closeList 表。openList 表由待考察的节点组成，closeList 表由已经考察过的节点组成。算法的详细流程在 Algorithm2 中给出。

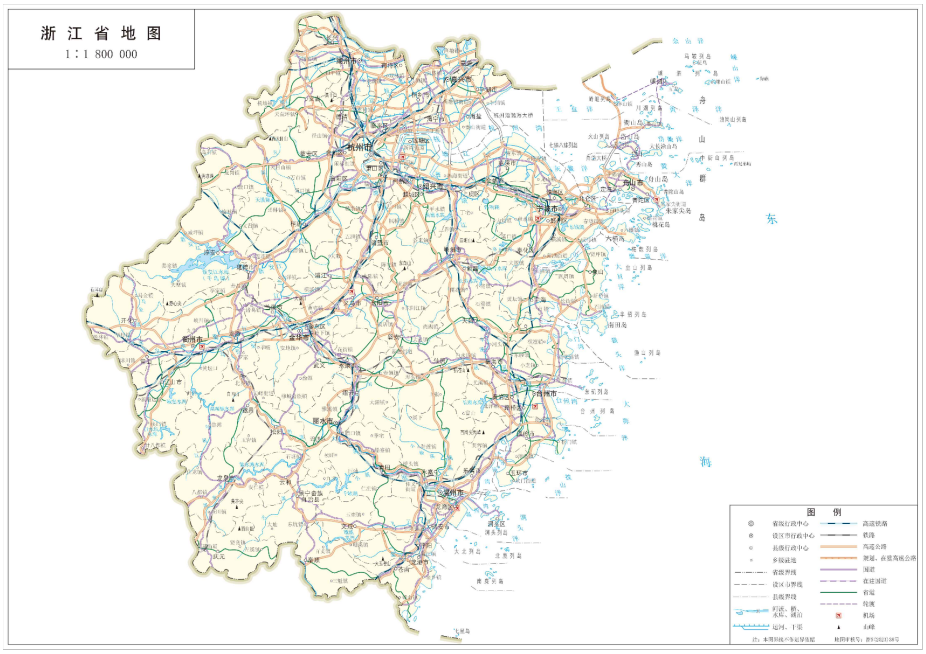

Figure 1: 浙江交通图

3 问题建模与求解

在本章节中，我们将描述所要解决的全局路径规划问题，并且给出模型。当我们打开地图，可以看到各点之间的路线是十分复杂的，如 Fig. 1 所示。我们以浙江省为例，可以看到各个地级市之间的交通网络是非常复杂的，包括铁路，公路和运河等等。如果我们建立一个详细的模型来描述它，这将会是一个复杂的过程。而且我们需要大量的数据来支撑我们的模型。因此，为了尽量贴近现实模型的请况且不至于使得模型复杂化，我们选取了各地级市和一部分县市作为我们连通图的节点，同时仅选用公路来作为各节点之间的边。

在这种情况下，我们可以给出我们简化后的模型。我们选取了 14 个市县来建模，各市县的交通路线仅选择国道或者高速公路，相邻节点路线长度用边的权重来表示。为了简化表示，边均用直线来表示。如此，我们便可以给出我们的简化模型，如 Fig. 2 所示。我们需要处理的问题就是，如何在给定起始点和终点的情况下，计算出两点之间的最优路径（距离最短），并给出这条路径。在本文中，我们采用 Dijkstra 算法和 A* 算法分别对该问题进行求解。

3.1 Dijkstra 算法求解

我们假设从衢州市出发到宁波市，使用 Dijkstra 算法求解起始点到各点的最小距离，并给出从起始点到终点的最优路径。我们把最终结果呈现在表中。最终回溯路径，得到从衢州到宁波的最优路径为衢州，建德，杭州，绍兴，宁波。

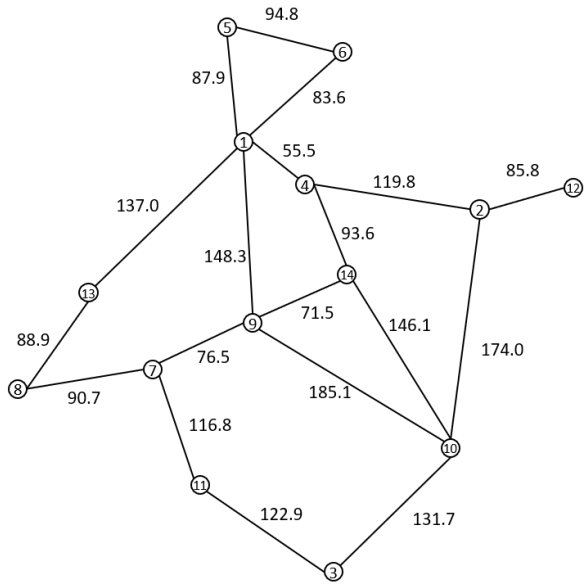

Figure 2: 简化模型

Table 1: 各点离起始点的最短距离

Position	Code	Minimum distance (km)
杭州	1	225.9
宁波	2	401.2
温州	3	330.4
绍兴	4	281.4
湖州	5	313.8
嘉兴	6	309.5
金华	7	90.7
衢州	8	0
东阳	9	167.2
台州	10	352.3
丽水	11	207.5
舟山	12	487
建德	13	88.9
嵊州	14	238.7

3.2 A* 算法求解

在使用 A* 算法求解问题时，需要选取合适的估计代价。常用的估计代价有曼哈顿距离和欧几里得距离，本文采用欧几里得距离，即两市之间的直线距离。在实际场景中，欧氏距离可以轻松得到，且该代价总是小于实际代价的，可以使 A* 算法搜索到最优解，因此我们选择欧式距离作为估计代价。最终使用 A* 算法求解得到的路径为衢州，建德，杭州，绍兴，宁波。可以看到与 Dijkstra 算法的求解结果一致，证明算法是有效的。

4 总结

在本文中，针对自动驾驶的全局路径规划，我们采用图论中的规划算法对问题进行了求解。首先，我们对现实地图进行了简化和提炼，抽象出了我们需要的模型。最后我们采用 Dijkstra 算法和 A* 算法对我们的路径规划问题进行了求解。

5 附录

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
#include <stack>
using namespace std;

#define INF 1e9

// Dijkstra算法函数
void dijkstra(vector<vector<float>>& graph, int start, vector<float>& distances, vector<int>&
     prenode) {
    int n = graph.size();
    vector<bool> visited(n, false); // 记录节点是否被访问过
    distances.assign(n, INF); // 初始化距离数组
    distances[start] = 0; // 起始节点到自身的距离为0
    prenode.assign(n, -1);
    // 使用优先队列实现堆优化,存储节点编号和距离
    priority_queue<pair<float, int>, vector<pair<float, int>>, greater<pair<float, int>>> pq;
    pq.push({0, start});
    // 开始迭代
    while (!pq.empty()) {
        float dist_u = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        // 如果当前节点的距离已经大于记录的最短距离,则忽略
        if (dist_u > distances[u]) {
            continue;
        }

        // 将该节点标记为已访问
        visited[u] = true;

        // 更新所有未访问节点的距离
        for (int v = 0; v < n; ++v) {
            if (!visited[v] && graph[u][v] != INF) {
                if (distances[u] + graph[u][v] < distances[v]) {
                    distances[v] = distances[u] + graph[u][v];
                    prenode[v] = u;
                    pq.push({distances[v], v});
                }
            }
        }
    }
}
```

```cpp
int main() {
    // 示例图的邻接矩阵表示
    vector<vector<float>> graph = {
        {0, INF, INF, 55.5, 87.9, 83.6, INF, INF, 148.3, INF, INF, INF, 137.0, INF},
        {INF, 0, INF, 119.8, INF, INF, INF, INF, INF, 174.0, INF, 85.8, INF, INF},
        {INF, INF, 0, INF, INF, INF, INF, INF, INF, 131.7, 122.9, INF, INF, INF},
        {55.5, 119.8, INF, 0, INF, INF, INF, INF, INF, INF, INF, INF, INF, 93.6},
        {87.9, INF, INF, INF, 0, 94.8, INF, INF, INF, INF, INF, INF, INF, INF},
        {83.6, INF, INF, INF, 94.8, 0, INF, INF, INF, INF, INF, INF, INF, INF},
        {INF, INF, INF, INF, INF, INF, 0, 90.7, 76.5, INF, 116.8, INF, INF, INF},
        {INF, INF, INF, INF, INF, INF, 90.7, 0, INF, INF, INF, INF, 88.9, INF},
        {148.3, INF, INF, INF, INF, INF, 76.5, INF, 0, 185.1, INF, INF, INF, 71.5},
        {INF, 174.0, 131.7, INF, INF, INF, INF, INF, 185.1, 0, INF, INF, INF, 146.1},
        {INF, INF, 122.9, INF, INF, INF, 116.8, INF, INF, INF, 0, INF, INF, INF},
        {INF, 85.8, INF, INF, INF, INF, INF, INF, INF, INF, INF, 0, INF, INF},
        {137.0, INF, INF, INF, INF, INF, INF, 88.9, INF, INF, INF, INF, 0, INF},
        {INF, INF, INF, 93.6, INF, INF, INF, INF, 71.5, 146.1, INF, INF, INF, 0},
    };

    int start = 7; // 起始节点编号
    int end = 1; // 结束节点编号
    stack<int> s; // 用于将回溯前驱点倒序
    vector<float> distances; // 存储最短距离的数组
    vector<int> prenode; // 存储每个节点的前驱节点
    // 调用Dijkstra算法计算最短路径
    dijkstra(graph, start, distances, prenode);

    // 输出结果
    cout << "Shortest distances from node " << start << " to other nodes:" << endl;
    for (int i = 0; i < distances.size(); ++i) {
        cout << "To node " << i << ": " << distances[i] << endl;
    }
    int k = end;//回溯路径
    cout << "Shortest route from start to end is: " << endl;
    while (k != -1)
    {
        k = prenode[k];
        if(k != -1) s.push(k+1);
    }
    while (!s.empty())
    {
        cout << s.top() << " -> ";
        s.pop();
    }
    cout << end+1;
    return 0;
}
```

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <cmath>
#include <climits>
#include <algorithm>
#include <unordered_map>
#include <chrono>
using namespace std;

#define INF 1e9

// A*算法函数
vector<int> AStar(vector<vector<float>>& graph, int start, int end, vector<vector<float>>
     estimate) {
    int n = graph.size();

    // 计算起始节点到目标节点的估计代价（这里简单地使用节点编号的差值作为估计代价）
    auto heuristic = [&](int node) {
        return estimate[node][end];
    };

    // 使用优先队列实现堆优化，存储节点和对应的f值（g值+估计代价）
    priority_queue<pair<float, int>, vector<pair<float, int>>, greater<pair<float, int>>> pq;
    pq.push({0, start});

    // 使用哈希表记录每个节点的父节点，用于最终回溯得到最短路径
    unordered_map<int, int> parent;
    parent[start] = -1;

    // 初始化距离数组
    vector<float> g(n, INF);
    g[start] = 0;

    // 开始迭代
    while (!pq.empty()) {
        int f = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        // 如果当前节点是目标节点，则结束搜索
        if (u == end) {
            break;
        }

        // 遍历当前节点的所有邻居节点
        for (int v = 0; v < n; ++v) {
            if (graph[u][v] != INF) {
                // 计算邻居节点的实际代价
                int cost = graph[u][v];
                int new_g = g[u] + cost;

                // 如果邻居节点没有被访问过或者新的实际代价更小，则更新信息并加入优先队列
                if (new_g < g[v]) {
                    g[v] = new_g;
                    parent[v] = u;
                    int h = heuristic(v);
                    pq.push({new_g + h, v});
                }
            }
        }
```

```
    }

    // 回溯得到最短路径
    vector<int> shortestPath;
    int current = end;
    while (current != -1) {
        shortestPath.push_back(current);
        current = parent[current];
    }
    reverse(shortestPath.begin(), shortestPath.end());

    return shortestPath;
}
int main() {
    // 示例图的邻接矩阵表示
    vector<vector<float>> graph = {
        {0, INF, INF, 55.5, 87.9, 83.6, INF, INF, 148.3, INF, INF, INF, 137.0, INF},
        {INF, 0, INF, 119.8, INF, INF, INF, INF, INF, 174.0, INF, 85.8, INF, INF},
        {INF, INF, 0, INF, INF, INF, INF, INF, INF, 131.7, 122.9, INF, INF, INF},
        {55.5, 119.8, INF, 0, INF, INF, INF, INF, INF, INF, INF, INF, INF, 93.6},
        {87.9, INF, INF, INF, 0, 94.8, INF, INF, INF, INF, INF, INF, INF, INF},
        {83.6, INF, INF, INF, 94.8, 0, INF, INF, INF, INF, INF, INF, INF, INF},
        {INF, INF, INF, INF, INF, INF, 0, 90.7, 76.5, INF, 116.8, INF, INF, INF},
        {INF, INF, INF, INF, INF, INF, 90.7, 0, INF, INF, INF, INF, 88.9, INF},
        {148.3, INF, INF, INF, INF, INF, 76.5, INF, 0, 185.1, INF, INF, INF, 71.5},
        {INF, 174.0, 131.7, INF, INF, INF, INF, INF, 185.1, 0, INF, INF, INF, 146.1},
        {INF, INF, 122.9, INF, INF, INF, 116.8, INF, INF, INF, 0, INF, INF, INF},
        {INF, 85.8, INF, INF, INF, INF, INF, INF, INF, INF, INF, 0, INF, INF},
        {137.0, INF, INF, INF, INF, INF, INF, 88.9, INF, INF, INF, INF, 0, INF},
        {INF, INF, INF, 93.6, INF, INF, INF, INF, 71.5, 146.1, INF, INF, INF, 0},
    };
    vector<vector<float>> estimate = {
        {0, 143.7, 258.0, 47.3, 86, 78.2, 116.5, 191.4, 108.8, 218.7, 201.2, 199.6, 116.5, 104.2},
        {143.7, 0, 224.5, 93.7, 180.7, 122.8, 203.1, 275.2, 138.7, 134.0, 220, 64.5, 220.6, 74.5},
        {258.0, 224.5, 0, 225.3, 326.4, 303.7, 158.2, 209.9, 150.2, 131.7, 92.4, 265.6, 215.5,
            175},
        {47.3, 93.7, 225.3, 0, 107.9, 81.3, 137.8, 201.6, 86.1, 174.4, 184.5, 159.2, 138.6, 57.4},
        {86, 180.7, 326.4, 107.9, 0, 66.9, 205.7, 246.4, 177.6, 278.6, 268.8, 221.8, 175.7,
            160.6},
        {78.2, 122.8, 303.7, 81.3, 66.9, 0, 214.7, 270.1, 168.4, 240.8, 265.9, 163.2, 200.9,
            131.1},
        {116.5, 203.1, 158.2, 137.8, 205.7, 214.7, 0, 77.5, 61.7, 178.1, 73.4, 268.2, 57.3,
            126.3},
        {191.4, 275.2, 209.9, 201.6, 246.4, 270.1, 77.5, 0, 139.6, 252.6, 118.6, 344.4, 70.1,
            204.0},
        {108.8, 138.7, 150.2, 86.1, 177.6, 168.4, 61.7, 139.6, 0, 134.9, 96.4, 204.9, 95.5, 66.4},
        {218.7, 134.0, 131.7, 174.4, 278.6, 240.8, 178.1, 252.6, 134.9, 0, 149.2, 165.2, 227.6,
            117.5},
        {201.2, 220, 92.4, 184.5, 268.8, 265.9, 73.4, 118.6, 96.4, 149.2, 0, 278.5, 130.4, 149.8},
        {199.6, 64.5, 265.6, 159.2, 221.8, 163.2, 268.2, 344.4, 204.9, 165.2, 278.5, 0, 288.3,
            141.5},
        {116.5, 220.6, 215.5, 138.6, 175.7, 200.9, 57.3, 70.1, 95.5, 227.6, 130.4, 288.3, 0,
            150.3},
        {104.2, 74.5, 175, 57.4, 160.6, 131.1, 126.3, 204.0, 66.4, 117.5, 149.8, 141.5, 150.3, 0},
    };

    int start = 7; // 起始节点编号
    int end = 1; // 结束节点编号
}
```

```
    // 调用A*算法计算最短路径
    vector<int> shortestPath = AStar(graph, start, end, estimate);
    // 输出最短路径
    cout << "Shortest path from node " << start+1 << " to node " << end+1 << ": ";
    for (int i = 0; i < shortestPath.size() - 1; i++) {
        cout << shortestPath[i]+1 << " -> ";
    }
    cout << shortestPath.back()+1 << endl;

    return 0;
}
```

References

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pp. 287–290, 2022.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.