开发者说丨Apollo项目Routing模块A*算法剖析

在图论中，应用最广泛的就是**搜索算法**了，比如，**深度优先搜索**、**广度优先搜索**等。然而在一些大型网游中需要你从当前位置走到某一目的地，而你只需要在地图中找到该目标位置，点击即可自动寻路走过去，那么这个自动寻路的功能是怎么实现的呢？

在实际场景中，游戏地图通常会很大，而且其中的路，障碍物等都不是很规则，并不能简单的将它们抽象成节点和线段。在寻路的过程中需要绕过所有障碍物，并且找到一条不会绕路的路径。似乎最短路是最符合要求的，但是如果地图很大，节点很多很多，那么求最短路的算法执行效率还是太低了。

不管是游戏中的寻路还是我们日常生活中地图软件的路线规划，其实都不需要找到那条最短路，而是在权衡效率和路线质量的情况下，找到一个次优解即可，**A* 算法**就是可以平衡效率和路线质量，找到次优路线的搜索算法。

A*（A-Star)算法是一种静态路网中求解最短路径最有效的直接搜索方法，也是解决许多搜索问题的有效算法。算法中的距离估算值与实际值越接近，最终搜索速度越快。

那么A* 算法在Apollo项目Routing模块中应用是怎样的呢？

以下，ENJOY

A*算法基本原理

广义上而言，**A*（A-Star)算法**就是一种动态规划算法，只不过A*算法在每步迭代计算时作出了更为严格的限制。关于动态规划，可以参考社区往期发布的文章**《开发者说丨动态规划及其在Apollo项目Planning模块的应用》**。

A*算法基于代价函数f(n)=g(n)+h(n)计算最短路径，其中f(n)是结点n的总代价函数，g(n)是从初始结点到结点n的移动代价，h(n)是从结点n到目标结点的启发代价。

如果不考虑具体实现代码，A*算法是相当简单的。有两个集合，**OPEN集**和**CLOSED集**。其中OPEN集保存待考察的结点。开始时，OPEN集只包含一个元素：初始结点。CLOSED集保存已考查过的结点。开始时，CLOSED集是空的。如果绘成图，OPEN集就是被访问区域的边境（Frontier），而CLOSED集则是被访问区域的内部（interior）。每个结点同时保存其父结点的指针，以便反向溯源。

在主循环中重复地从OPEN集中取出最好的结点nf值最小的结点）并检查之。如果n是目标结点，则我们的任务完成了。否则，从OPEN集中删除结点n并将其加入CLOSED集。然后检查它的邻居n’。如果邻居n’在CLOSED集中，表明该邻居已被检查过，不必再次考虑（若你确实需要检查结点n’的g值是否更小，可进行相关检查，若其g值更小，则将该结点从CLOSED集中删除）；如果n’在OPEN集中，那么该结点今后肯定会被考察，现在不必考虑它。否则，把它加入OPEN集，把它的父结点设为n。到达n’的路径的代价g(n’)，设定为g(n) + movementcost(n, n’)。

下面是算法伪代码：

```
1   OPEN = priority queue containing START
2
3   CLOSED = empty set
4
5   while lowest rank in OPEN is not the GOAL:
6
7     current = remove lowest rank item from OPEN
8
9     add current to CLOSED
10
11    for neighbors of current:
12
13      cost = g(current) + movementcost(current, neighbor)
14
15      if neighbor in OPEN and cost less than g(neighbor):
16
17        remove neighbor from OPEN, because new path is better
18
19      if neighbor in CLOSED and cost less than g(neighbor): **
20
21        remove neighbor from CLOSED
22
23      if neighbor not in OPEN and neighbor not in CLOSED:
24
25        set g(neighbor) to cost
26
```

```
        add neighbor to OPEN

        set priority queue rank to g(neighbor) + h(neighbor)

        set neighbor's parent to current

reconstruct reverse path from goal to start by following parent pointers

(**) This should never happen if you have an admissible heuristic.
However in games we often have inadmissible heuristics.
```

<左右滑动以查看完整代码>

Apollo项目Routing模块 A*算法源码剖析

下面直接贴出Apollo项目Routing模块A*算法的源码，在相关代码处添加注释。

A*策略类

```
class AStarStrategy : public Strategy {
 public:
  explicit AStarStrategy(bool enable_change);
  ~AStarStrategy() = default;
  // A*搜索函数
  virtual bool Search(const TopoGraph* graph, const SubTopoGraph* sub_graph,
                      const TopoNode* src_node, const TopoNode* dest_node,
                      std::vector* const result_nodes);

 private:
  // 清空上次结果
  void Clear();
  // 计算src_node到dest_node的启发式代价
  double HeuristicCost(const TopoNode* src_node, const TopoNode* dest_node);
```

```
    // 计算结点node到终点的剩余距离s
    double GetResidualS(const TopoNode* node);
    // 计算边edge到结点to_node的剩余距离s
    double GetResidualS(const TopoEdge* edge, const TopoNode* to_node);

  private:
   // 允许变换车道
    bool change_lane_enabled_;
    // OPEN集
    std::unordered_set<const TopoNode*> open_set_;
    // CLOSED集
    std::unordered_set<const TopoNode*> closed_set_;
    // 子父结点键值对
    // key: 子结点
    // value: 父结点
    std::unordered_map<const TopoNode*, const TopoNode*> came_from_;
    // 移动代价键值对
    // key: Node
    // value: 从源结点移动到Node的代价
    std::unordered_map<const TopoNode*, double> g_score_;
    // 结点的进入距离键值对
    // key: Node
    // value: Node的进入距离
    std::unordered_map<const TopoNode*, double> enter_s_;
};
```

<左右滑动以查看完整代码>

待考察结点

```
struct SearchNode {
    const TopoNode* topo_node = nullptr;
    double f = std::numeric_limits<double>::max();

    SearchNode() = default;
    explicit SearchNode(const TopoNode* node)
        : topo_node(node), f(std::numeric_limits<double>::max()) {}
    SearchNode(const SearchNode& search_node) = default;
    // 重载<运算符，改变小于逻辑，以更其作为优先级队列std::priority_queue的内部元素时，
    // std::priority_queue的栈顶元素永远是值为最小的一个。
    bool operator<(const SearchNode& node) const {
      // in order to let the top of priority queue is the smallest one!
      return f > node.f;
    }

    bool operator==(const SearchNode& node) const {
```

```
17        return topo_node == node.topo_node;
18      }
19    };
```

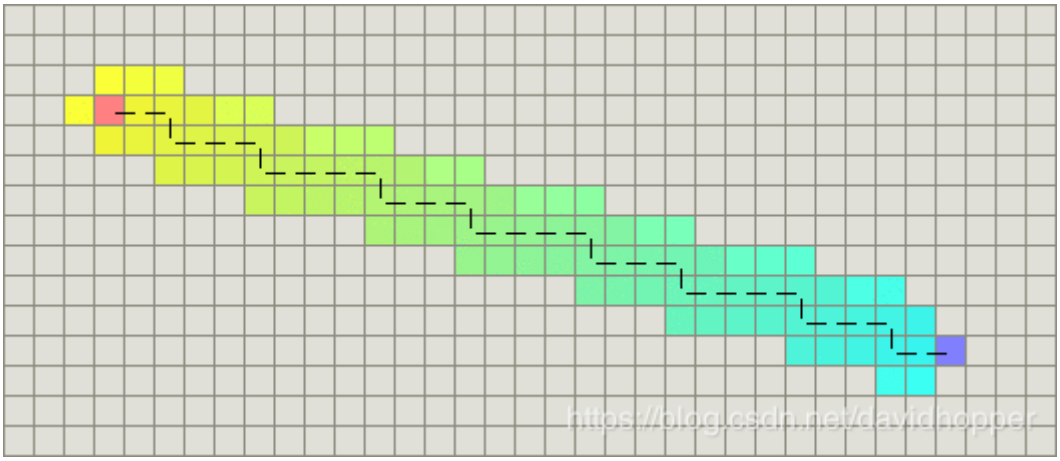
<左右滑动以查看完整代码>

到目标结点的启发式代价计算方法

标准的启发式代价函数是曼哈顿距离（Manhattan distance）。所谓曼哈顿距离就是两点在南北方向上的距离加上在东西方向上的距离。曼哈顿距离又称为出租车距离，曼哈顿距离不是距离不变量，当坐标轴变动时，点间的距离就会不同。Apollo项目采用的就是曼哈顿距离函数：

```
1        H(n) = abs ( n.x – goal.x ) + abs ( n.y – goal.y )
```

<左右滑动以查看完整代码>




```
1   double AStarStrategy::HeuristicCost(const TopoNode* src_node,
2                                       const TopoNode* dest_node) {
3     const auto& src_point = src_node->AnchorPoint();
4     const auto& dest_point = dest_node->AnchorPoint();
5     double distance = fabs(src_point.x() - dest_point.x()) +
6                       fabs(src_point.y() - dest_point.y());
7     return distance;
8   }
```

<左右滑动以查看完整代码>

A*算法

```
bool AStarStrategy::Search(const TopoGraph* graph,
                           const SubTopoGraph* sub_graph,
                           const TopoNode* src_node, const TopoNode* dest_node
                           std::vector* const result_nodes) {
  Clear();
  AINFO << "Start A* search algorithm.";

  // std::priority_queue是一种容器适配器，它提供常数时间的最大元素查找功能，
  // 亦即其栈顶元素top永远输出队列中的最大元素。但SearchNode内部重载了<运算符，
  // 对小于操作作了相反的定义，因此std::priority_queue的栈顶元素
  // 永远输出队列中的最小元素。
  std::priority_queue open_set_detail;
  // 将源结点设置为检查结点
  SearchNode src_search_node(src_node);
  // 计算检查结点的启发式代价值f
  src_search_node.f = HeuristicCost(src_node, dest_node);
  // 将检查结点压入OPEN集优先级队列
  open_set_detail.push(src_search_node);
  // 将源结点加入OPEN集
  open_set_.insert(src_node);
  // 源结点到自身的移动代价值g为0
  g_score_[src_node] = 0.0;
  // 设置源结点的进入s值
  enter_s_[src_node] = src_node->StartS();

  SearchNode current_node;
  std::unordered_set<const TopoEdge*> next_edge_set;
  std::unordered_set<const TopoEdge*> sub_edge_set;
  // 只要OPEN集优先级队列不为空，就不断循环检查
  while (!open_set_detail.empty()) {
    // 取出栈顶元素（f值最小）
    current_node = open_set_detail.top();
    // 设置起始结点
    const auto* from_node = current_node.topo_node;
    // 若起始结点已抵达最终的目标结点，则反向回溯输出完整的路由，返回。
    if (current_node.topo_node == dest_node) {
      if (!Reconstruct(came_from_, from_node, result_nodes)) {
        AERROR << "Failed to reconstruct route.";
        return false;
      }
      return true;
    }
    // 从OPEN集中删除起始结点
    open_set_.erase(from_node);
```

```
// 从OPEN集队列中删除起始结点
open_set_detail.pop();

// 若起始结点from_node在CLOSED集中的计数不为0，表明之前已被检查过，直接跳过
if (closed_set_.count(from_node) != 0) {
  // if showed before, just skip...
  continue;
}
// 将起始结点加入关闭集
closed_set_.emplace(from_node);

// if residual_s is less than FLAGS_min_length_for_lane_change, only move
// forward
// 获取起始结点from_node的所有相邻边
// 若起始结点from_node到终点的剩余距离s比FLAGS_min_length_for_lane_change要短，
// 则不考虑变换车道，即只考虑前方结点而不考虑左右结点。反之，若s比
// FLAGS_min_length_for_lane_change要长，则考虑前方及左右结点。
const auto& neighbor_edges =
    (GetResidualS(from_node) > FLAGS_min_length_for_lane_change &&
     change_lane_enabled_)
        ? from_node->OutToAllEdge()
        : from_node->OutToSucEdge();
// 当前测试的移动代价值
double tentative_g_score = 0.0;
// 从相邻边neighbor_edges中获取其内部包含的边，将所有相邻边全部加入集合：next_edge_set
next_edge_set.clear();
for (const auto* edge : neighbor_edges) {
  sub_edge_set.clear();
  sub_graph->GetSubInEdgesIntoSubGraph(edge, &sub_edge_set);
  next_edge_set.insert(sub_edge_set.begin(), sub_edge_set.end());
}
// 所有相邻边的目标结点就是我们需要逐一测试的相邻结点，对相结点点逐一测试，寻找
// 总代价f = g + h最小的结点，该结点就是起始结点所需的相邻目标结点。
for (const auto* edge : next_edge_set) {
  const auto* to_node = edge->ToNode();
  // 相邻结点to_node在CLOSED集中，表明之前已被检查过，直接忽略。
  // 注意：这里和A*算法伪代码中的片段：
  // if neighbor in CLOSED and cost less than g(neighbor):
  //    remove neighbor from CLOSED
  // 有所差异。
  // 似乎修改为if (closed_set_.count(to_node) > 0)更好?
  if (closed_set_.count(to_node) == 1) {
    continue;
  }
  // 若当前边到相邻结点to_node的距离小于FLAGS_min_length_for_lane_change，表明不能
  // 通过变换车道的方式从当前边切换到相邻结点to_node，直接忽略。
  if (GetResidualS(edge, to_node) < FLAGS_min_length_for_lane_change) {
    continue;
  }
  // 更新当前结点的移动代价值g
  tentative_g_score =
      g_score_[current_node.topo_node] + GetCostToNeighbor(edge);
```

```
      // 如果边类型不是前向，而是左向或右向，表示变换车道的情形，则更改移动代价值g
      // 的计算方式
      if (edge->Type() != TopoEdgeType::TET_FORWARD) {
        tentative_g_score -=
            (edge->FromNode()->Cost() + edge->ToNode()->Cost()) / 2;
      }
      // 总代价 f = g + h
      double f = tentative_g_score + HeuristicCost(to_node, dest_node);
      // 若相邻结点to_node在OPEN集且当前总代价f大于源结点到相邻结点to_node的移动代价g，表明现有
      // 从当前结点到相邻结点to_node的路径不是最优，直接忽略。
      // 因为相邻结点to_node在OPEN集中，后续还会对该结点进行考察。
      // open_set_.count(to_node) != 0修改为open_set_.count(to_node) > 0似乎更好
      if (open_set_.count(to_node) != 0 &&
          f >= g_score_[to_node]) {
        continue;
      }
      // if to_node is reached by forward, reset enter_s to start_s
      // 如果是以向前（而非向左或向右）的方式抵达相邻结点to_node，则将to_node的进入距离更新为
      // to_node的起始距离。
      if (edge->Type() == TopoEdgeType::TET_FORWARD) {
        enter_s_[to_node] = to_node->StartS();
      } else {
        // else, add enter_s with FLAGS_min_length_for_lane_change
        // 若是以向左或向右方式抵达相邻结点to_node，则将to_node的进入距离更新为
        // 当前结点from_node的进入距离加上最小换道长度，并乘以相邻结点to_node长度
        // 与当前结点from_node长度的比值（这么做的目的是为了归一化，以便最终的代价量纲一致）。
        double to_node_enter_s =
            (enter_s_[from_node] + FLAGS_min_length_for_lane_change) /
            from_node->Length() * to_node->Length();
        // enter s could be larger than end_s but should be less than length
        to_node_enter_s = std::min(to_node_enter_s, to_node->Length());
        // if enter_s is larger than end_s and to_node is dest_node
        if (to_node_enter_s > to_node->EndS() && to_node == dest_node) {
          continue;
        }
        enter_s_[to_node] = to_node_enter_s;
      }
      // 更新从源点移动到结点to_node的移动代价（因为找到了一条代价更小的路径，必须更新它）
      g_score_[to_node] = f;
      // 将相邻结点to_node设置为下一个待考察结点
      SearchNode next_node(to_node);
      next_node.f = f;
      // 当下一个待考察结点next_node加入到OPEN优先级队列
      open_set_detail.push(next_node);
      // 将to_node的父结点设置为from_node
      came_from_[to_node] = from_node;
      // 若相邻结点不在OPEN集中，则将其加入OPEN集，以便后续考察
      if (open_set_.count(to_node) == 0) {
        open_set_.insert(to_node);
      }
    }
  }
```

```
  // 整个循环结束后仍未正确返回，表明搜索失败
  AERROR << "Failed to find goal lane with id: " << dest_node->LaneId();
  return false;
}
```