一份详细的工作总结

YWJ

2018年4月11日

1 introduction

本文结构如下:

1. 全文总览, 以及比赛历程回顾.
2. 详细的的问题场景描述, 关于寻路算法 A-Star 应用于问题场景的分析.
3. 基于代价传播(Price Disseminating)的寻路算法, 以及其应用于问题场景的分析.
4. 利用有向空间的特性加速迪杰斯特拉的方法(Oriented Space Dijkstra), 利用GPU加速有向空间迪杰斯特拉搜索的方法(GPU-OSD), 处理起点终点在空间中坐标不定的办法, 以及上述手段应用于问题场景的分析.
5. V3.X(PD)工程文档.
6. V6.X(GPU-OSD)工程文档.
7. 附录.

在17年11月份我开始着手这个项目. 问题分为两部分, 一是根据公司的提供的数据训练学习器, 用以预测未来较大范围内的天气情况; 二是根据预测的得到的天气地图规划无人飞行器的路线. 我主要负责第二部分的路线规划算法, 简单的理解是把它当作一个带障碍的寻路问题(天气恶劣的点当作障碍), 已有各种版本实现且原理简单的A-star算法可以很好的解决带障碍寻路. 但经过两次简短的讨论后, 我们决定放弃A-star, 这其中的考虑这里先不谈.

与此同时另一个原理复杂得多, 但显然更适合路线规划的寻路算法(PD)逐渐孕育成型, 就一些细节问题讨论完善后, 我总结了PD代价正向反向传播应当遵循的几条原则, 并给出的伪代码. 在11月底12月初, 我做出了V1.0的实现版本, 搜索耗时将近两个小时, 提交结果说明我们的寻路算法是可靠的. 但在这次提交后V1.0很快就被放弃掉了, 采用CPU并发的V2.X版本将搜索速度提高到了25分钟左右. 在12月底完成的PD算法最终版V3.X利用阙值记录文件将搜索速度提升至12分钟左右.

但PD算法的速度和开销仍不能让我们满意, 甚至对我们自己对问题的最初理解也产生了怀疑. 将原问题当成带障碍寻路问题最不可解释的地方在于障碍阙值的选取, 即多坏的天气应当被当作障碍, A-star和PD这一类带障寻路算法从原理上无法克服这一问题. 最终我们想到了概率, 即不泾渭分明的确定哪些点安全哪些点危险, 而是将每个点的天气映射成安全或危险概率. 这样障碍阙值的问题就不存在了. 但这实际上将原问题变成了一个带权重的路径搜索问题, Dijkstra是我们第一时间想到的办法. 对此我完成了V4.0版本, 但这一版本并未投入使用. 地图结点数量在十亿数量级, 尽管是稀疏图, Dijkstra为了找到全

局最优解, 每个结点必须出入队列一次, 这导致V4.0的运行时间需要48小时之久. 而由于Dijkstra的一些特性, 没法在算法内部使用并发, 这些问题一度使得使用概率模型代替确定性模型的想法被搁置.

在1月份经过一阵的挣扎后, 我意识到了两个可能提速Dijkstra的因素. 一是如果搜索地图的结点是分层次的, 那么就不需要耗费开销维持Dijkstra的队列有序, 而且任意地图通过增加时间维总能变成可分层图. 二是这种可分层的图上的Dijkstra算法能使用GPU加速. 为此我完成了V5.X版本, 利用笔记本上的1050Ti, 算法搜索完一天的地图仅需12秒, 完成全部五天的搜索只需一分钟. 至此我们的寻路算法已趋于成熟.

进入2月复赛后, 塞题发生了重大变化, 飞行器的起飞时间不再固定, 且互相不能冲突需要规划. 为了找到全局最优解, 不得不考虑一架飞行器所有可能起飞的时间点, 这使得搜索空间增大了两个数量级. 最终算法V6.X版本解决了这个问题,在GPU-OSD的逐层搜索中,只保留唯一的最优上游节点,这使得到达每个点的最优路径和起点时间坐标都被记录在传播后的地图中. 针对起飞时间冲突, 我设计了一个简单的退避算法来避免冲突产生,尽管这不是一个全局最优的规划, 但在实践中我们发现冲突产现象并不多见, 且这种退避算法能很好的解决少量冲突. V6.X总搜索耗时约2分钟左右.

2 问题场景与A-Star算法

赛题链接, 天气预测问题与路线规划问题将被分开介绍.

2.1 天气预测部分

根据英国气象局提供的真实天气测量数据, 预测大范围的天气地图.

1. 气象指标. 初赛中天气只考虑风速指标, 默认飞行器进入风速大于15的区域会坠毁, 但由于预测总体偏差, 这个阙值并不可靠. 复赛中增加了降雨指标, 默认降雨大于5会导致飞行器坠毁, 风速规定不变.

2. 地图规格. 赛方提供了一张548×421大小的气象数据图, 飞行器需两分钟从任意点行至其上下左右方向的相邻点.

3. 气象数据. 气象数据有效时间为一小时, 赛方提供了每天3-21时的气象数据, 总计五天. 在某天的任意小时, 地图上每个点据提供了十个特征, 按赛方说法, 这些特征本身就是气象局的十个不同模型给出的当前区域降雨预测值(0-47.1), 准确率在95%上下. 实则数据异常值较多, 根据统计, 异常点(十个预测器均认为安全训练集却标记为危险的点)数量在10%以上. 根据可视化分析, 初赛数据集存在明显的混乱, 每天的第6和8, 10和12小时的训练特征和标签是完全颠倒的. 只在前六小时的数据上和在后12小时上用相同方法训练的学习器,前者指标(标准差)明显更低(少了一个数量级). 可惜当我们发现这一点时初赛已经结束. 幸运的是复赛训练数据并备有发现类似问题.复赛的降雨数据形式与风速类似.

2.2 路线规划部份

每天3点钟10架推进式无人运输飞行器将开始从伦敦海德公园飞往英国其他10个目的地城市, 总计五天. 11个城市在地图上的坐标均已给出. 每架飞机只能支撑18小时飞行时间, 超时或进入危险区域即判定坠毁. 得分方式为: 所有未坠毁飞行器的路线时长+坠毁飞行器*1440. 总共需要在五张气象图上规划50条飞行路线. 进入复赛后. 赛题产生了很大的变化, 首先是不再规定飞行器的起飞时间(初赛必须在三点整起飞), 但所有飞行器必须在21点前到达, 否则算作坠毁; 其次是所有飞行器的起飞间隔必须大于10分钟, 否则当次提交成绩无效, 按所有路线坠毁算作72000分.

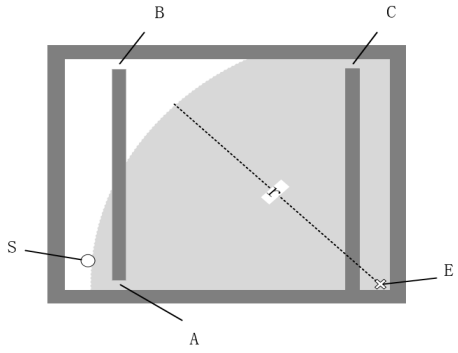

图 1: 起点为S, 终点为E, 灰色区域的所有点到终点的距离都小于r, 如果按到终点距离最短原则搜索, 则B点将在A, C两点之后被搜索到, 由于A, C两点已经构成一条到达终点的通路, 在搜索到B之前算法已经退出了. 但$S \to B \to C \to E$显然比$S \to A \to C \to E$更优.

路线规划的难点在于动态的天气变化, 每天18个小时的天气各不相同, 因此不在二维空间中规划路线, 而是加入时间轴(后面都记作z轴), 在三位空间中完成规划是理所当然的想法. 这其实使搜索空间从一个二维的无向空间变成了三维的有向空间, 也为后面的Dijkstra加速算法带来了意想不到的可能性. 但这也会产生问题, 即终点的坐标不再确定了, 根据路线长度不同, 到达终点时终点的z坐标是不确定的. 经过复赛改题后, 连起点的坐标也不再确定了(不再默认三点整出发). 这些问题如何解决这里暂且不谈.

2.3 适用于带障碍寻路问题的A-Star算法

所谓障碍寻路, 可以形象的理解为走迷宫. 一张地图上只有两类点, 墙或者路. 另外两个特殊的路点被标记为起点和终点, 算法需要找到一条最短的路径连通起点终点, 或是判定终点不可达. 广为人知的A-Star算法为这类问题提供了一个高效的解决方案. 它的优点很明显, 原理十分简单, 易于实现, 效率比深搜广搜高出很多, 比蚁群算法的计算资源需求少得多, 一块cpu就能在短时间内搜索问题场景级别的地图, 在算法外部实现并发也相当容易. 为了准确的描述PD算法在问题场景下相对于A-Star的优越性, 这里先说明A-Star的原理.

A-star可以看作广度或深度优先搜索的一种推广, 它们的区别在于选取搜索搜索点的方式不同, 广度优先总是优先搜索离起点曼哈顿距离近的点, 深度优先则优先扩展所有可达点中离起点曼哈顿距离最远的点. A-star则同时考虑到终点和起点的距离, 比如优先扩展到终点和起点曼哈顿距离之和最小的点. 当然, 距离的度量方式有很多, 还可以有欧氏距离, 对角线距离等等, 但实际应用中以曼哈顿距离居多. 需要指出的是, 只考虑到终点的距离最短扩展是不能保证搜到的路径一定最短的, 比如图1.

1. 针对不同的终点, A-Star必须执行多次搜索. 因为针对不同的终点, 搜索点到终点的距离不同, 扩展点的选取也会不同. 即针对不同的终点的搜索互相独立. 这很容易用并发解决, 但也导致了计算资源的极大浪费, 在GPU-OSD中, 这个问题得到了解决, 只需搜索地图一次, 所有可达点的最优路径都能被确定.

2. A-Star针对每对确定的起点终点, 只返回某一条确定的最短路径. 显然一对起点终点间存在多条不同的最短路径, 它们之间孰优孰劣尚需规划, 但A-Star杜绝了任何规划的可能. 这在PD算法中的得到了解决, PD能以小于等于A-Star的代价, 标记出起点终点间所有可能的最短路径. 而对于最后用到的概率模型, 这个问题本身就不存在.

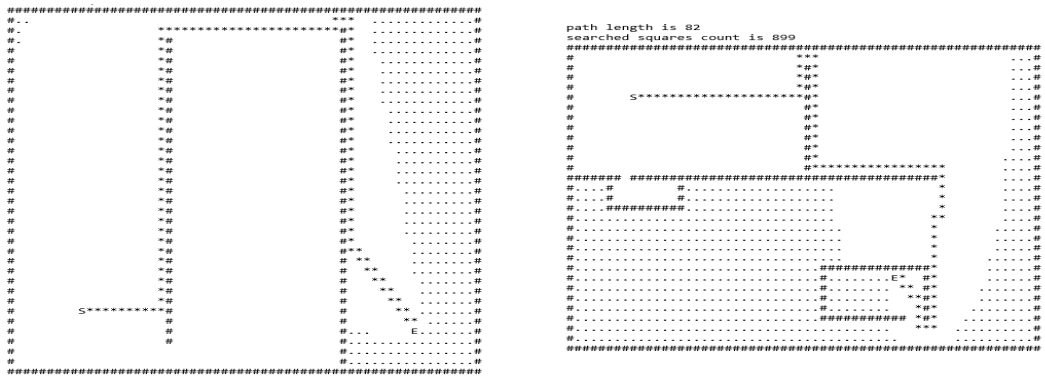

图 2: 两张地图上的A-Star搜索. 注意A-Star给出的路线总是沿墙的.

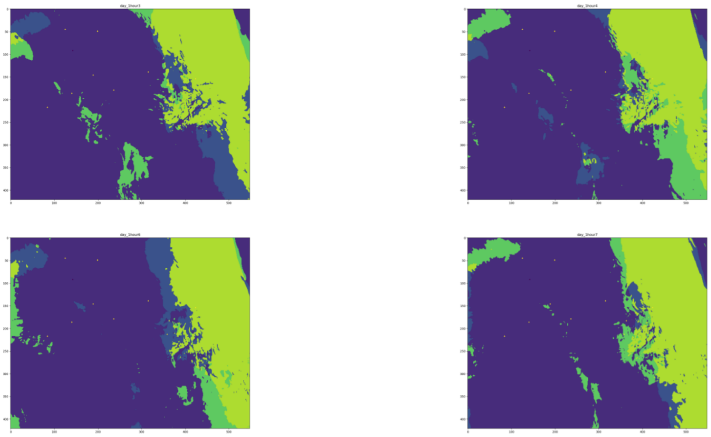

图 3: 第二天的某四个小时气象图: 颜色越深的区域越安全, 学习器判为安全而实际上也安全(深紫色)>学习器判为危险而实际上安全(深蓝色)>学习器判为危险而实际上危险(黄绿色)>学习器判为安全而实际上危险(亮绿色). 亮绿色的区域是导致路线死亡的主因, 而亮绿色几乎都附着在黄绿色(墙)的边沿.

3. A-Star需要位置一个有序队列或堆来存储所有可扩展点. 这同传统的Dijkstra一样, 大量的计算开销被用来维持所有可扩展点的优先顺序. 在PD和GPU-OSD中, 没有任何需要维持有序的队列或堆.

4. A-Star的扩展原则决定了搜索出的最优路径只有遇到墙点才会改变方向. 如果前进方向上没有墙点, A-Star会根据扩展原则一直延保证距离和不变的方向搜索, 直到遇上连续的墙点才被迫改变方向. 图2展示了这种情况. 这是A-star应用在此问题下最致命的弱点, 基于宏观自然界的数值都是连续变化的认识, 有理由相信与风速较大的点相邻的点风速同样较大, 在此问题下场景下, 所有墙点的周边很可能是潜在的墙点, 或者说越靠近越危险. 而A-Star始终贴墙走. 正是基于这一认识, 我们一开始就放弃了A-Star, 后面的PD算法很好的解决了这个问题, 而在概率模型中, 这个问题如第二点一样不成立.

基于以上认识, 我们意识到A-Star的有效性必须依赖天气地图预测的绝对准确, 这显然是办不到的. 事实上再初赛中由于数据本身的问题, 预测器效果极差, 大量的危险区域被判做安全, 如图3. 需要澄清的是, 上两张图或下两张图的第一二类错误区域看起来是正好相反的, 这并非巧合或是我们的可视化代码有问题, 而是由于数据本身的标签在某些小时上对调过, 如前文已经指出的, 这些对调每天都有, 且发生在固定的小时. 不知官方是否有意为之. 需要再次强调的是, 复赛数据没有发现任何类似的异常.

3 确定性模型上的改进算法PD

3.1 路由器寻址算法——洪泛法

洪泛法是经典的内部网关协议. 若路由器A希望查询到未知路由B的最短路径和距离, A向所有相邻路由广播查询请求, 收到查询的路由若不是B且路由表里没有记录B, 则继续递归地广播下去. 直到某一路由返回B的信息, 收到回复的路由需向所有向其发起过请求的路由回复查询, 以供它们确定到达B的下一跳地址. 洪泛法收敛后, 网络上的所有节点都得知了到达B的最短路径.

代价传播算法可以类比洪泛法, 但与之正好相反, 我们希望所有终点都得知自己到起点的最短路径(起点只有一个, 而终点有十个), 而非从起点开始发起查询. 于是算法从起点开始广播自己的位置, 而非查询请求. 收到广播的点能以此确定自己到达起点的距离以及最短路径, 进而继续递归地广播自己的位置. 最终所有的终点都能确定自己到达起点的最短路径. 在一次搜索下可以确定到达所有终点的最短路径, 而非A-Star下需对不同重点进行独立搜索.

3.2 代价传播算法

PD算法分为两个过程, 前向传播和反向寻路.

前向传播是扩散代价的过程, 代价的定义是由起点到终点的最短路径距离相比理想距离的差值. 以曼哈顿距离为例, 终点的代价=起点到终点的最小路径长度-起点到终点的曼哈顿距离. 在传播代价时按代价递增做广度优先搜索, 搜索到终点后, 继续搜索完所有和终点代价相等的点. 至此, 算法的前向搜索部分已经完成, 到达所有等于或低于终点代价的点的所有最短路径都已经被标记在地图中了. 单次最大传播距离是确定路径的单位最大单位距离, 例如在迷宫问题下, 一次只能沿坐标移动一格, 以曼哈顿距离度量为例, 单次最大传播距离应该取1, 若允许斜向运动, 单次最大传播距离应该取2(斜向运动一次曼哈顿距离变化为2, 沿坐标轴的传播距离仍为1), 代价前向传播需要遵循下面一条原则:

$$\text{代价变化量} = \text{单次最大传播距离} - \text{到起点的距离变化量}$$

在这条原则下, 代价的前向传播过程满足下面几条性质:

1. 某一点相对起点的代价如果是确定的, 那么能据此确定它所有相邻点相对起点的代价.
2. 若点B的代价由点A传播而来, 那么称A是B的父节点. 任意点的父节点可能存在多个, 且由不同的父节点传播而来的代价总是相等的.
3. 若A是B的父节点, 那么B的代价总是大于或等于A的代价.
4. 若A是B的父节点, 那么路径$A \to B$一定在起点到B的最短路径上.
5. 起点到点A的最短路径长度 = 起点到点A的度量距离 + 点A的代价.

反向寻路是根据标记后的地图生成最短路径的过程. 由性质4, 只要有终点开始递归地寻找当前点的父节点, 一定可以形成一条终结于起点的最短路径. 由于任意点的代价可能由不同的父节点传播而来, 最短路径可能存在多条. 由性质2可知, 只要满足代价传播公式的相邻点都可以作为当前点的父节点. 且由于性质3, 一次前向搜索后, 所有等价的最短路径都被标记在地图上了, 因为终点所有祖先节点的代价均已被标记.

如在一个简单的迷宫中前向传播完成后得到的图4. 起点S到终点E, F的所有最短路径都被标记在代价传播图中了. 根据代价传播公式, E可以是F的父节点, 但F不能作为E的父节点, 显然S到达E的路径若经过了F就一定长于最短路径.

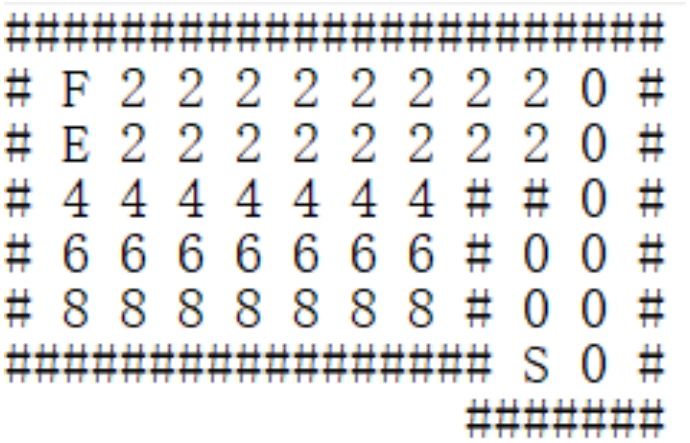

图 4: 一个简单场景下的代价传播图. 起点为S, 终点为E, F. 只能沿上下左右方向运动, 距离度量采用曼哈顿距离, 这张图下单次最大传播距离为1. 注意所有代价大于或等于4的点都与寻路无关, 在工程中不会为了标记这些点浪费计算资源, 它们为了展示代价传播规则才被标记在图中.

Algorithm 1: 二维地图上的代价传播

输入: 地图$A(x, y)$, 起点$s = (x_0, y_0)$, 终点表E, 安全阈值G

输出: 代价标记地图$B(x, y)$

init $B(:,:) \leftarrow -1$,$B(x_0, y_0) \leftarrow 0$,$Queue \leftarrow [s]$,$Queue_{next} \leftarrow []$
while E中有点未标记代价 **do**
  while $Queue$非空 **do**
    $p \leftarrow Queue.pop()$
    for q *in* p的相邻点 **do**
      if $B(q) == -1$ *and* $A(q) < G$ **then**
        $B(q) \leftarrow B(p) + price(s, p, q)$
        if $Bq == Bp$ **then**
          $Queue.push(q)$
        else
          $Queue_{next}.push(q)$
        end
      else
        continue
      end
    end
  end
  $Queue \leftarrow Queue_{next}$
  $Queue_{next} \leftarrow []$
end
Function $price(s, p, q)$
  return $distance_{min} - distance(s, q) + distance(s, p)$

Algorithm 2: 反向寻路

输入: 地图$A(t,x,y)$,代价标记地图$B(t,x,y)$, 起点s,终点e

输出: 最优路径P

```
init P ← [e],p ← e
while s! = p do
    选择p的相邻点中代价小于或等于p的任意点q
    P.append(q)
    p ← q
end
```

3.3 PD应用于问题场景

PD应用与赛题的问题场景下相比A-star有许多优势.

1. 搜索速度主要以算法找到最短路径前搜索过的点占地图纵面积比例来衡量, 在单条路径搜索上, PD与A-Star在这个指标上是相当的, 但PD搜索每个点的代价比A-Star小得多, 前面已经提过, A-Star需要在所有可扩展点中优先扩展距离度量最小的点, 因此在每搜索一个点后加入一些新的可扩展点时, 不得不重新对所有的可扩展点排序; 而PD不许要维持这样一个有序的堆或队列, 每次搜索一个点只需简单的四则运算. 若算上多条路径搜索, PD的优势就非常明显了, PD只需搜索地图一次而A-Star是十次.

2. 前文已经提到过A-Star的贴墙问题, PD算法下可以借二次规划解决这个问题. 由于PD标记了所有的最短路径, 这些路径之间是有优劣之分的. 在实际工程中, 我们使用简单的贪心法来二次规划最短路径, 若某个节点有多个父节点, 总是选取气象指标最安全的父节点作为它的最短路径上游节点. 这种简单的做法在实践中相当有效.

3. 对确定性模型, 障碍阙值的选择始终是个问题. 在V3.X中, 搜索地图并不使用固定的阙值, 而是从初始阙值开始, 针对每条路径根据给定步长反复调整阙值. 在一天的地图上, 若某条路线无法找到通路, 那么降低阙值继续搜索; 反之如果搜索到了通路, 那么增加阙值再次搜索, 直到无法搜出路径, 取所有搜索出的路径中阙值条件最大的作为最终路径. 前者可以保证算法一定能对所有路径给出答案, 后者保证搜索出的路径上所有的点风速都低于某一值. 由于贪心法不是全局最优的规划, 找到路径上所有点的风速都低于某一最小阙值的唯一路径提供了一定程度的全局规划能力.

4. PD使用了阙值记录文件来确定么条路径的初始搜索阙值, 如果一次搜索比阙值记录文件的步长精度更高, 那么PD会用最终每条路径确定的阙值更新记录文件, 下次搜索时各条路径都从记录文件中读取自己对应的初始值, 而非从15.0开始搜索.

4 使用GPU加速Dijkstra

4.1 使用概率模型替代确定性模型

前面已经提过, 确定性模型最致命的问题在于阙值的选择. 尽管应用中的PD设计了一套机制来自动选择阙值, 但问题并没有被彻底解决. 首先算法只能在很小的范围内调整阙值, 参考官方给出的阙值15.0, 假设采用0.1的调整步长, 暴力搜索到14.0就需要10轮. 其次阙值的调整步长精度无法确定, 在实际运行中, 我们发现步长精度并非越高越好, 且步长每低一个数量级搜索代价就要增大十倍, 因此PD只能在很小的范围内调整步长. 总而言之, 我们无法为阙值的选择找到任何理论依据, 这是确定性模型中注定缺失的一环.

相比于确定性模型, 我们假设在地图的任意一点飞机均已一定的概率坠毁, 而非在一些点上绝对安全或者一定危险. 这样做的好处有三:

1. 我们的学习算法不可能绝对准确, 学习器预测为安全的点并不一定安全, 引入坠毁概率可以软化这些错误, 即不无条件地相信学习器.

2. 保留了预测值的相对大小信息. 在确定性模型中, 所有安全或危险的点都是等价的, 但预测器针对每个点给出的预测实际上有大小之分的, 或者说在所有的安全点中, 一些点实际上比另一些更安全, 但确定性模型将这些信息全都放弃了. 在概率模型中, 只要找到映射使学习器预测风速值更大的点坠毁概率更大, 这些大小信息将成为寻路的主要依据.

3. 使用概率统计模型不需要强行确定阈值. 前面已经说过, 我们无法为阈值的确定找到任何理论依据, 搜索阈值代价很大而作用有限. 而在概率统计模型中, 只要找到一个映射, 能将风速投影到值为$(0,1)$的概率区间, 并能保持任意两点间的预测大小关系, 这样的映射就是可行的. 实际上一个简单的线性分段函数就能完成这个任务.

4.2 概率模型简介

在已知所有点的坠毁概率的前提下, 我们希望路线到终点的期望得分最小. 期望得分由路径坠毁概率, 坠毁惩罚时间和路径耗时算得. 为了形式化的描述, 先约定如下记号:

1. $P(A)$为起点到点A的路径坠毁概率, $p(A)$为点A的坠毁概率.

2. $E(A)$为起点到点A的得分期望, $L(A)$为路径A的时间开销, 考虑题目给出的坠毁惩罚为1440, 那么有:

$$E(A) = L(A)[1 - P(A)] + 1440P(A)$$

我们的目标是找到期望得分最小的路径, 这实际上是一个超大规模的动态规划问题. 但我们发现了如下事实: 如果某一点A的$E(A)$确定, 对任意可有A到达的B, 如果$p(B)$已知, 那么$E(B)$可由以求得. 推导如下:

$$\begin{aligned}
&P(B) = 1 - [1 - P(A)][1 - p(B)] \\
&E(B) = (L(A) + 2)[1 - P(B)] + 1440P(B) \\
&= (L(A) + 2)[1 - P(A)][1 - p(B)] + 1440\{1 - [1 - P(A)][1 - p(B)]\} \\
&= 1440 + (L(A) + 2)[1 - P(A)][1 - p(B)] - 1440[1 - P(A)][1 - p(B)] \\
&= 1440 + 2[1 - P(A)][1 - p(B)] + L(A) \cdot [1 - P(A)][1 - p(B)] - 1440[1 - P(A)][1 - p(B)] \\
&= 1440 + 2[1 - P(A)][1 - p(B)] + [1 - p(B)]\{L(A) \cdot [1 - P(A)] - 1440 + 1440P(A)]\} \\
&= 1440 + 2[1 - P(A)][1 - p(B)] - 1440[1 - p(B)] + [1 - p(B)]\{L(A) \cdot [1 - P(A) + 1440P(A)]\} \\
&= 1440 + 2[1 - P(A)][1 - p(B)] - 1440[1 - p(B)] + [1 - p(B)]E(A) \\
&\Rightarrow E(B) = 2[1 - P(A)][1 - p(B)] + 1440p(B) + [1 - p(B)]E(A)
\end{aligned}$$

实际上我们的问题转化成了正好对应迪杰斯特拉搜索的情况. 最短路径问题变成了最小期望得分问题, A, B点间的边长由$P(A), p(B)$共同决定, 而在路径$A \to B$下, $E(B)$总可由$E(A)$和A, B间边长经上式计算而来.

4.3 Dijkstra慢在哪里?

Dijkstra的主要开销在于维持候选点集合有序, 即每次从候选点集中选择总路径最短的点, 再将与所选点相邻而不在候选集中, 且未被选中过的点加入集合, 并不破换集合按选取优先级的排序. 但我们发现在具有层次结构的图中, 维持有序结构是不必要的. 我们给层次结构图下的定义如下: 图的所有节点可被划分成若干非空集合, 满足如下性质:

1. 各个集合之内的点无法互相连通.
2. 所有集合有唯一排序, 使得在该排序下: a). 序号相邻的集合中的节点有连通, 且方向只能由低序号集合指向高序号集合. b). 序号不相邻的集合之间无法互相连通.

上面两条性质可能不具有画面感, 直观地讲, 带有层次结构的图只能由一批节点到达另一批节点, 无法返回, 也无法在某批节点内部传播. 树或森林都满足上面两个性质, 属于层次结构图, 不具备循环结构或层内计算结构的神经网络计算流图也是层次结构图.

Dijkstra中, 每当一个节点的最短路径被确定时, 所有与之相邻的点中, 当前路径长度大于它的点的路径可能发生变化, 当前路径长度小于它的点的路径不会变化, 因为这些点的最短路径不可能经过一个最短路径长度比它当前路径长度还大的点. 因此dijkstra必须每次选择当前路径长度最小的点, 这也决定了dijkstar难以并发, 因为一旦图和起点给定, dijkstra的搜索序列就确定了, 一个搜完才能搜下一个. 但在层次图中, Dijkstra不必按最小原则选取候选点(或者说这已经不是Dijkstra了), 因为某个点的最短路径确定后, 只有下一层中由它可达的点的路径才有可能发生变化. 换言之, 由于某一层中所有点的最短路径只与上一层有关, 只要逐层确定层次图的最短路径, 在某一层中的候选点的选择顺序是无关紧要的, 并发也相当容易.

而且我们注意到, 只要给搜索地图添上时间维度, 任意的有向图都可以转化为层次结构图. 如图5., 记一次转移到相邻点的过程时间开销为1. 这种转化的结果一定满足上述层次结构的两条性质, a). 层内无连接, 不可能由一个点转移至相邻点而不消耗时间. b). 相邻层只能单向连通, 因为时间不能回溯. c). 没有跨层连通, 因为时间只能按单位时间流逝.

重要的是, 一旦图给定, 各层之间的连通关系都是确定且相同的, 这为并发提供了基础. 在二维地图寻路问题中, 这种连通关系更加简单, 因为一个点只能在单位时间内移动到它上下左右四个方向上的相邻点, 或者保持原地不动(由于问题场景中天气会随时间浮动, 因此保持原地不动在一些情况下是有意义的). 由于层间的连通结构一致, 且层内每个点与下一层的连通结构也相同, 这决定了我们的算法可以在GPU上并发.

4.4 在GPU上实现加速

在贴出算法之前先形式化问题场景. 地图是三维矩阵(t, x, y), 记为M, 在问题场景中, 单位时间为2分钟, 最长时限为18小时, 因此$t = 540$, x, y由二维地图尺寸决定.

由于传播到任意层的路径长度是相等的(t值相等), 算法只需根据前一层的最小路径坠毁概率推算出后一层的最小坠毁概率, 由于后一层中除边界以外的任意节点在前一层中方存在5个可能的父节点, 算法通过平移前一层对应矩阵来对齐两层, 每次计算一个方向上的传播而来的坠毁概率, 五个方向对应的坠毁概率存在$temp$中,$temp$沿第0维按最小值降维, 就得到了下一层的最小坠毁概率, 而$temp$在第0维的降维坐标标

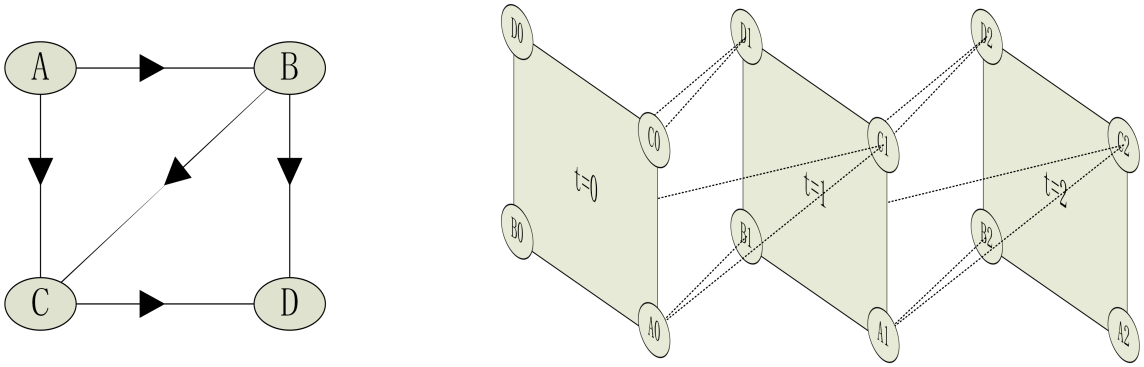

图 5: 左边的有向图由于$B \to C$的存在不能划分为层次结构, 如假设每次节点转移耗时为1, 能得到右边的层次结构图.

Algorithm 3: GPU搜索

输入: 地图$M(t,x,y)$, 起点(x_0,y_0)

输出: 路径标记地图$A(t,x,y)$: 记录到任意节点最短路径上 ,它的父节点相对于它的方向

最小期望的分地图$B(t,x,y)$: 记录到达任意点的最小期望得分.

最小坠毁概率地图$C(t,x,y)$: 记录到达任意点的最小坠毁概率

最短路径值地图$D(x,y)$: 记录任意点的最小期望得分路径长度, 据此可以知达到任意点的最小期望路径终点在A中的t坐标

init $A[:,:,:] \leftarrow -1, B[0,:,:] \leftarrow 7200, B[0,x_0,y_0] \leftarrow 0, C[0,:,:] \leftarrow 1, C[0,x_0,y_0] \leftarrow 0$
for $i \in range(1,t)$ **do**
  $A[i,:,:], C[i,:,:] \leftarrow spread(M[i,:,:], C[i-1,:,:])$
  $B[i,:,:] \leftarrow 1440 * C[i,:,:] + i * 2 * (1 - C[i,:,:])$
end
$D \leftarrow min\_reduce(B)$
Function $spread(M[i-1,:,:], C[i-1,:,:])$
  init $temp[5,x,y]$ **for** $d \in [0(stay), 1(left), 2(right), 3(up), 4(down)]$ **do**
    $m \leftarrow move(M[i-1,:,:], d)$
    $c \leftarrow move(C[i-1,:,:], d)$
    $temp[d,:,:] \leftarrow 1 - (1 - C[i-1,:,:])(1 - M[i,:,:])$
  end
  $C[i,:,:], A[i,:,:] \leftarrow min\_reduce(temp)$
  return $A[i,:,:], C[i,:,:]$
Function $move(M,d)$
  if $d == 0$ **then**
    return M
  else
    M 沿 d 方向平移一个下标, 超出矩阵范围的部分删除, 空出的部分用1补全
    return M
  end

记了后一层中的最小坠毁概率路径由前一层传播而来的方向. 在最后的期望得分地图中, 以同一点为终点但在不同时间到达的路径会有不同的期望得分, 同样将期望得分地图按时间维做最小值降维, 得到二维地图上所有点的最小期望得分.

4.5 GPU-OSD应用于问题场景

比赛后期我们的领先成绩主要依赖于GPU-OSD算法, 如不考虑GPU的计算瓶颈, 算法的时间复杂度为$O(t)$. 实际工程依赖Pytorch调用GPU, 在笔记本的1050Ti上, 整个工程从读取地图到写完提交表格仅需12秒, 提交结果表明, 概率模型下的全局最优解远胜于确定性模型, 新模型和算法的理论基础也十分牢固, 几乎找不到难以解释或可以完善的地方, 至此, 我们的搜索算法基本定型. 复赛中由于赛题改动, 我为GPU-OSD添加了一些机制以适应出发时间不固定的新环境, 算法的开销基本没有变化. 另外为了避免出发时间冲突, 在搜索同一天不同路径的时候维持一张冲突表以防止两家飞机起飞时间过近, 如检测到出发时间冲突, 退避算法会在已经标记好的地图中查找两条冲突路线中代价较小的次优解, 直到冲突表恢复, 这不是一个全局最优的规划, 因为较小代价的次优解可能与其它路线冲突而造成更大的损失, 但实际中发生二次冲突的情况很少, 退避算法从来没有在一天的搜索中检测到三次冲突, 这与赛题的场景有关, 对于十多个小时的起飞窗口, 间隔10分钟十个架次的确显得过于稀疏, 对于少量的一二次冲突, 上面简陋的退避算法已经能很好的应对了.

对于不确定的起飞时间, 只需在初始化的时候将起点在t维度上的所有点全部初始化为起点, 在传播过程中任意路径可能由起点在某个时刻的镜像点传播而来, 但算法只记录路径的最小坠毁概率, 最终依靠路径标记地图总能反推出最小期望路径起点的t坐标. 稍有不同的是不能再根据当前层的t值和最小坠毁概率直接计算得分期望了(由于出发时间可能不同, 同层的最小期望路径长度不再相等), 因而对于起飞时间不确定的状况, 算法实际传播层间的最小得分期望而非路径坠毁概率.

Algorithm 4: 全局退避

输入: 路径标记地图$A(t,x,y)$, 最小期望的分地图$B(t,x,y)$, 路径冲突表s

输出: 无冲突路径表s

```
while s存在冲突 do
    选择最先检测到的两条冲突路径(p, E(p)), (q, E(q))
    在B(t,x,y)中查找p, q对应终点的次优路径(p', E(p')), (q', E(q'))
    if E(p') − E(p) > E(q') − E(q) then
        用(q', E(q')替换s中的(q, E(q))
    else
        用(p', E(p')替换s中的(p, E(p))
    end
end
```