**算法模块**

以下是对模拟导航系统所使用的核心算法的介绍与说明，具体代码参见工程目录中algorithm 文件。

**一、地图初始化以及存储结构**

**1.1 数据结构详解**

本功能由mapsave实现，建立边表信息 edge[] 以及点表：mapPoint 存储该点链接的边与点，采用邻接表的形式存储点边对应关系，使用自写的标点、标边工具，对接腾讯地图api，在点表以及边表中录入北京邮电大学本部及沙河校区的地图信息。

下面将以简单的三个点来说明建图步骤：

点图的数据结构：

export interface mapPoint {

    id: string

    name: string

    position: {

        lat: number

        lng: number

    }

    neighborWalk: neighbor[]

    neighborBike: neighbor[]

}

首先对每个点进行id编号，id唯一，将对应的逻辑点的名字name 存储进入，并根据腾讯地图api提供的经纬度（lat、lng）录入信息，根据传回的逻辑信息修改该点的逻辑信息表。

点的邻接表里面的边分为两类：neighborWalk 以及neighborBike,对应该边是行走边或自行车边：

export interface neighbor {

    toPointId: string

    edgeId: string

}

此对象中toPointId 对应该边所指向的另一个点的id，edgeId存储该边详细信息的边（edge对象）的ID，通过edgeId寻找边表中的详细信息。此时边表中新建的边指向id为2的点，该边对应的边id为“1”，更详细信息见下表：

export interface edge {

    id: string

    length: number

    /\*\* range: 0-1 \*/

    congestionDegree: number

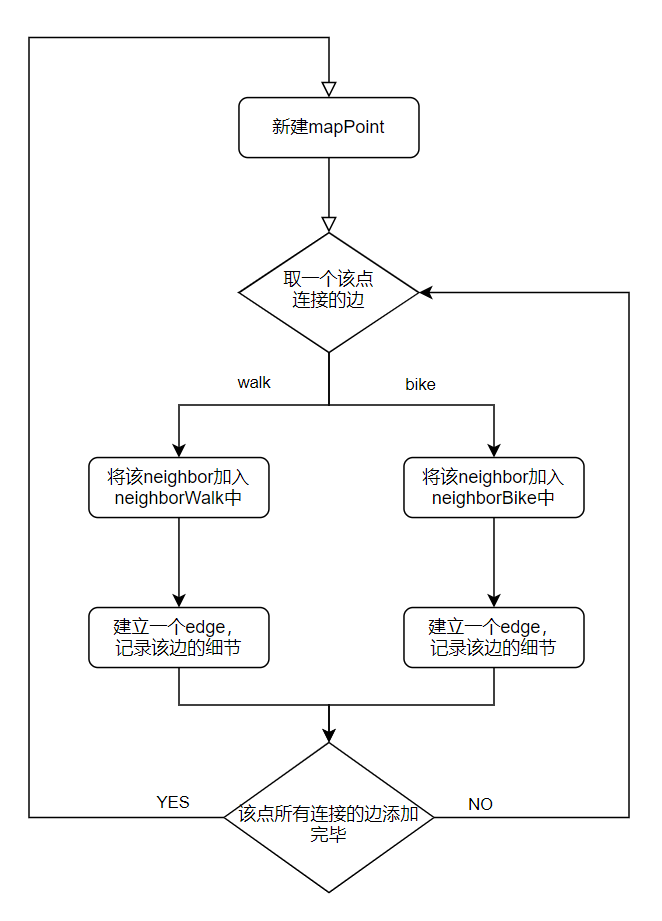
    /\*\* walk: 1, bike: 2, bus: 3 \*/

    type: TRANSPORT

    time: number }

此对象中记录了该边的id，以及该边长度length，该边拥挤度congestionDegree，该边交通工具类型type，该边需要的时间time。

对于图中的每一个点和边，我们都以此方式进行录入，最终北京邮电大学导航地图的数据信息录入完成。



建图流程图

**1.2 数据结构性能分析**

为了对地图存储结构的效率进行分析，假设共拥有N个节点,M条边，依据北京邮电大学的地图结构，M与N的关系可以解释为：

北京邮电大学双校区总体节点N\_bupt的值大于200，此时我们运用稀疏图/稠密图判断公式：

可得北京邮电大学地图结构为稀疏图，相比需要存储空间大小的邻接矩阵，空间大小近似为的邻接表更适合北京邮电大学的地图结构。

对于本次的地图结构算法所需的操作，可以总体归纳为对于图遍历，读取路径的操作，对于无向图的遍历操作，一次遍历总共的访问次数为 N + 2M。

**二、导航算法**

根据导航所需功能不同，我们分别采取了：A\*寻路算法，Dijkstra寻路算法，剪枝优化的多必经点寻路算法以及基于模拟退火优化的多必经点寻路算法，分别对应最短距离的导航情况，最短时间的导航情况、含有交通工具最短时间导航情况。以及多必经点的导航情况。

**2.1.1 最短距离导航算法（Astar算法）**

Astar是优化的Dijkstra，所以本篇先讲解Dijkstra作为前置，由于Dijkstra算法为数据结构基础算法，在此不再赘述，对该算法的详细描述可参考维基百科词条，以下仅对Dijkstra进行简单的描述。

Dijkstra作为单源最短路算法，可以在的时间内计算出从某个起点到其他点的最短路径，其算法核心为：

维护两个集合：已算出集合A和未算出集合B，其中起点初始默认在已算出集合A内。

维护一个dis[]数组，用来存储从起点到某个点的距离，除起点外其他初始化为正无穷。

在未算出集合中寻找dis最小的点S，加入已算出集合A，此时加入的点的邻居点P有三种情况：

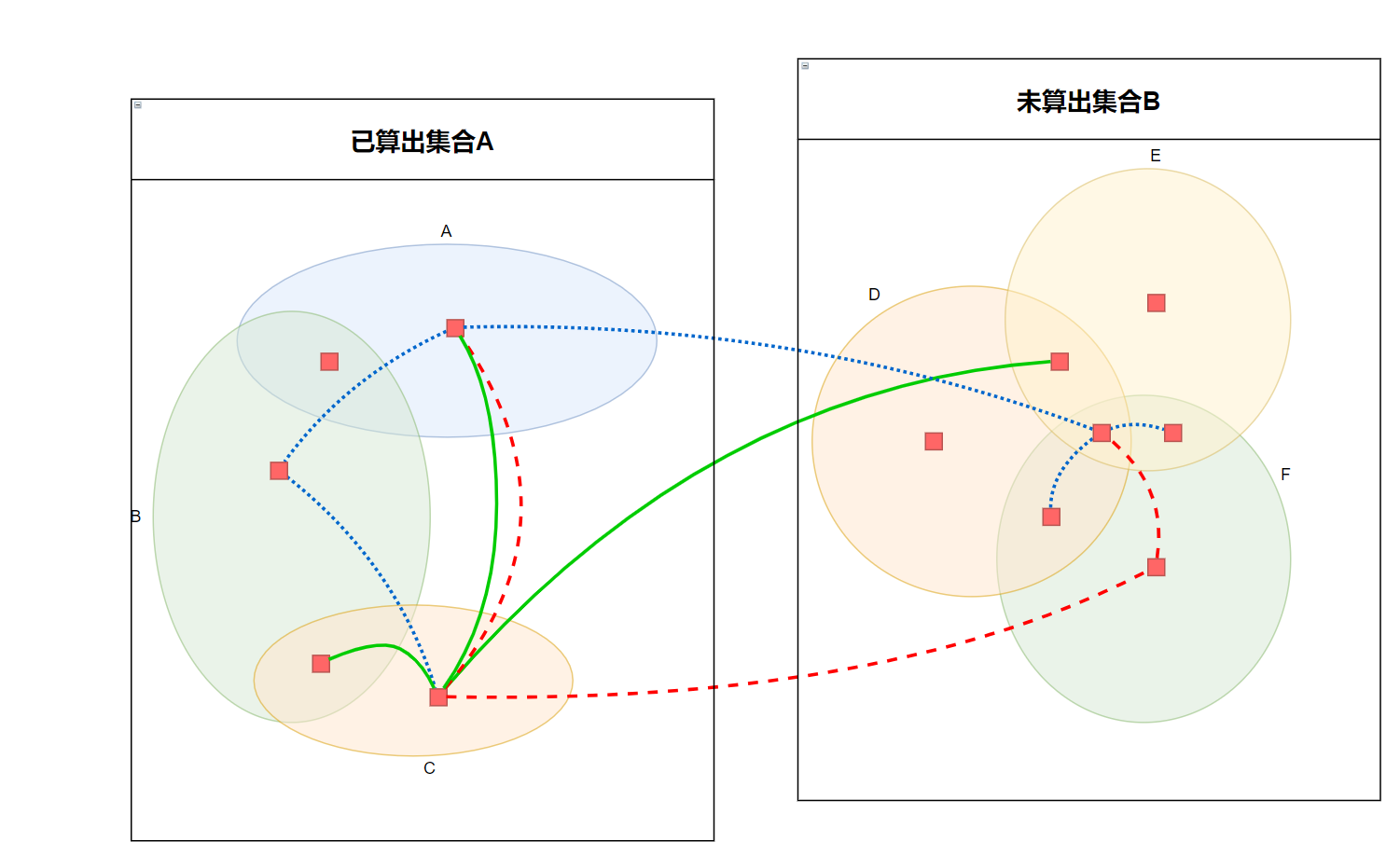
1.P已加入已算出集合A，此时不管

2.P未加入已算出集合A，也未接壤任何已算出集合A的点，此时更新其dis为：加入点的dis+加入点到这个邻居的边距离。

3.P未加入已算出集合A，但接壤了已算出集合A的点，dis已更新：取原值和新加入点的dis+加入点到这个邻居的边的距离的最小值（松弛）：

一次遍历（所有点加入已算出集合A）后，从起点到所有点的最小值都会记录到dis数组内，如果而记录路径只需维护该点从哪里来的father。

Dijkstra算法保证能找到一条从初始点到目标点的最短路径，只要所有的边都有一个非负的代价值。

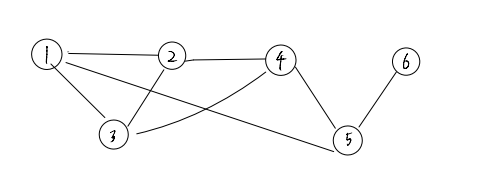


Astar算法建立在Dijkstra的基础上，是一种启发式搜索。与选择离初始结点最近的结点不同的是，它选择离目标最近的结点。它比Dijkstra算法快的多，因为它用了一个启发式函数（heuristic function）快速地导向目标结点。例如，如果目标位于出发点的南方，Astar将趋向于导向南方的路径。

在讨论A\*的标准术语中，g(n)表示从初始结点到任意结点n的代价，h(n)表示从结点n到目标点的启发式评估代价（heuristic estimated cost）。当从初始点向目标点移动时，A\*权衡这两者。每次进行主循环时，它检查未算出集合B中f(n)最小的结点n，其中f(n) = g(n) + h(n)，并将其加入已算出集合A，当终点加入已算出集合A时，停止算法计算。

我们可以发现，在每秒步速相同的情况下，作为一个基于现实世界地图制成的图结构，在未加入点dis相同的情况下，选择离终点近的点的优先级更高。

如下图：假如起点为1，终点为6，那么对于2和3，我们将优先选择点2作为下一个加入的点。



样例

所以我们使用的启发策略是基于曼哈顿距离的计算：

对这里我们用该点到终点的横坐标纵坐标差的和来计算H,通过dis的公式计算G，最后通过F = G + H 的策略使搜索方向始终以终点优先。

这里我们会发现，因为步速相同，边速相同，我们以终点方向为优先，A\*算法内部发生的事情是：在每一结点它都计算f(n) = g(n) + h(n)。当h(n)精确地和g(n)匹配时，f(n)的值在沿着该路径时将不会改变。不在最短路径上的所有结点的f值均大于正确路径上的f值。如果已经有较低f值的结点，A\*将不考虑f值较高的结点，因此它肯定不会偏离最短路径。

**2.1.2 Astar算法性能分析**

与Dijkstra算法相比，A\*是启发性的搜索算法，其是更有目的性地(依赖启发函数的指引)向目的节点逼近，到达目的地后立刻剪枝。而Dijkstra算法则不同，它是盲目性地按算法规则一步步寻找下去。因此，Dijkstra算法是适合单源到其他所有节点的最短路径的搜索算法，从搜索的结果路径来看，Dijkstra算法从源节点到任何节点的路径肯定都是最短的，而A\*却未必，因为A\*的启发函数只是预估、估算出代价罢了，在步速相同、边速相同的情况下通过曼哈顿距离计算基本为最短路径（出问题的情况参见最短时间模块）。

从时间复杂度上看，Dijkstra算法的时间复杂度，如果是任何两点间的最短路径的话，算法时间复杂度为，而A\*则要好很多，从上面算法思想中可以大约看出，时间代价最大的是在每循环遍历处理所有节点环节上，假设点的平均出度为P，则两点间的A\*寻路时间复杂度的最坏情况可以近似为：

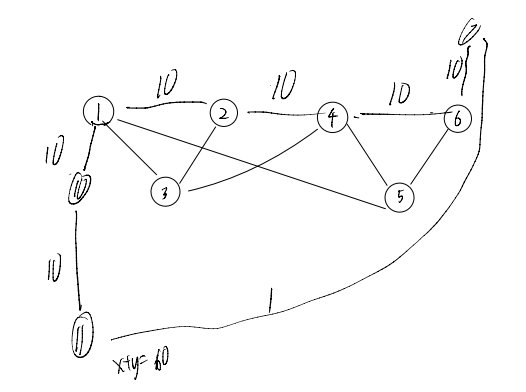
因此，时间复杂度为渐近，另外，由于剪枝步骤的存在，则A\*的每次寻路是随时都有可能提前结束的，结束的期望为 ，所以时间复杂度的期望为：

**2.2.1 最短时间导航算法（包含交通工具）（Dijkstra算法）**

若有认真阅读A\*部分说明，必能发现我们反复强调A\*优化的一个前提条件：

现实地图每秒步速相同的情况下

那么在引进拥挤度和自行车道的概念后，此优化方案便会出现漏洞。自行车道和速度不一的步行道会在某种情况下推翻上述“必然最短时间”的情况，如下例所示，我们计算从起点1到终点 7 的最短时间路径：



图中边数字为此边所需时间

其中11 - 7是只需要花费1分钟的自行车道。

此时按照A\*算法，会选择：1 2 4 6 7

但最优解确实从起点向下，走 10 11 7 然后走自行车道到达终点7

这种情况下采用A\*的启发式搜索会发生错误，所以我们在这种情况下将算法退化成单纯的Dijkstra的算法。

**2.2.2 Dijkstra算法性能分析**

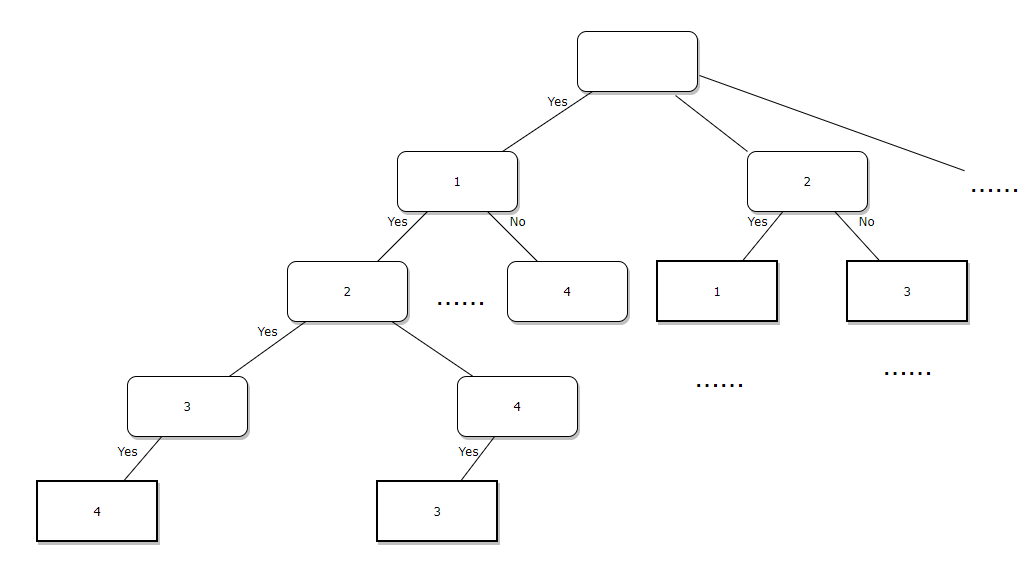
对于北京邮电大学具有N个顶点和M条边的带权有向图，Dijkstra算法的主循环体需要)时间。这个循环需要执行n-1次，所以完成循环需要时间。算法的其余部分所需要的时间不超过 ，总体时间复杂度为，使用堆优化的情况下，时间复杂度为。

相比时间复杂的弗洛伊德算法，Dijkstra拥有更小的时间复杂度，而基于Bellman-Ford优化的SPFA时间复杂度优化的不稳定性以及SLF、LLL优化构造较为困难，使得Dijkstra成为我们最好的选择。

**2.3.1 途径点算法（分支界限法）**

对于多必经点的要求，最简单的方法遍是枚举必经点的所有排列组合的可能性，进行枚举，此时的复杂度为 ,在7个点之后计算时间超过1s，这个时间复杂度是我们无法接受的。

运用分支界限法，我们很容易能证明一个结论：当枚举了一个情况后算出值MIN后，枚举新的情况，未将所有必经点枚举出来但已经超过MIN时便无需继续枚举下去，而当最终结果小于MIN时，便刷新MIN。这种剪枝在8个点的情况下相比纯暴力优化了0.5s。



如图所示，设必经点集合 = {1,2,3,4},当MIN为 1 2 3 4 算出来的值，而 计算 1 4发现已经超过所需的值时，后面所有的子情况无需计算，因为必定超过MIN的值，此为分支界限法的剪枝。

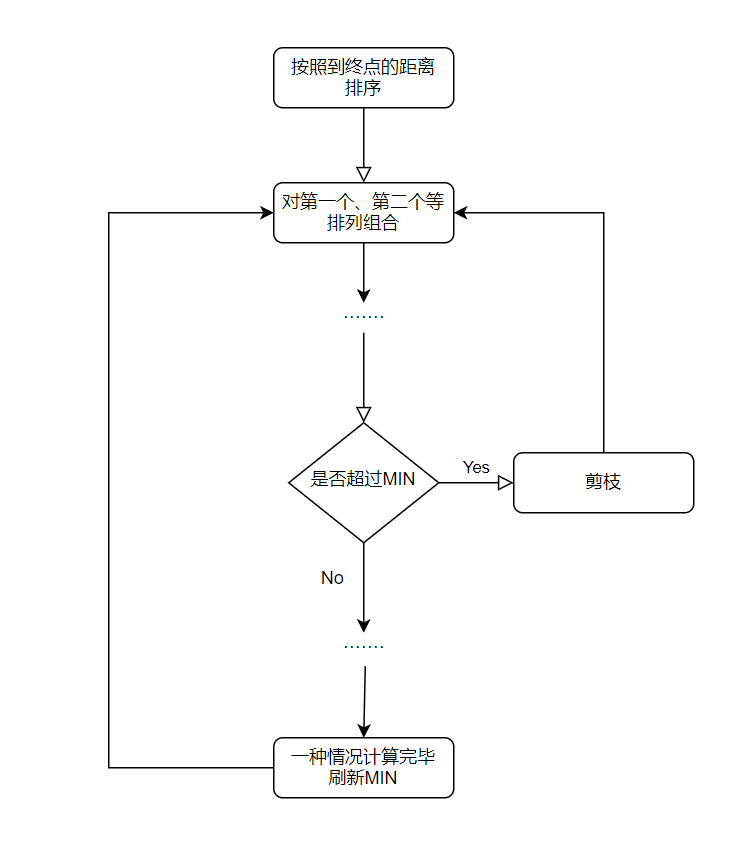
而我们又可以根据A\*的思想证明新的结论：排列组合中，必经点离起点越近，排在前面的可能性越高（自行车道、拥挤度不一的步行道会导致不确定性），此时算出来的值是最小值的可能性也越高，那么我们以必经点离起点近为优先级排序，进行排列组合，后续的枚举的情况被剪枝的概率也越高。

举个例子，如果有5个必经点，按照离起点的距离排序为 1 2 3 5 4

那么 1 2 3 5 4为顺序算出来的大概率是最小值，我们按优先级第一个搜它，之后在N!以1 2 3 5 4 为基础进行遍历。

当枚举到情况： 5 3 {子集} 时这种情况， 5 3就被剪枝了的概率十分大

而5 2{…}， 4 1{…}等这些根据优先级比较靠后的情况被剪枝的概率更高。



分支界限法流程

**2.3.2 途径点算法（分支界限法）性能分析**

经过计算，暴力枚举的时间复杂度T如下：

在超过7个点的情况下运行时间已超过1s，8个点时间为6s，9个点时间约为1分钟，超过9个点我们停止了测试。

在加入分支界限后，8个点的情况下相比暴力减少了0.5s，加入排序优化后，8个点的情况优化了0.3s，这两种优化属于剪枝，并没有对算法进行时间复杂度的优化，很明显的是，复杂度排列组合的暴力枚举算法是我们着重需要优化的对象，我们便引入了模拟退火算法。

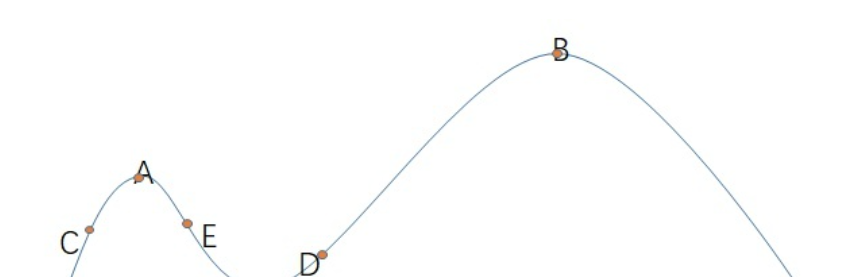
**2.4.1 途径点算法（模拟退火）**

模拟退火算法(Simulated Annealing,SA)最早的思想是由N. Metropolis等人于1953年提出。1983年,S. Kirkpatrick等成功地将退火思想引入到组合优化领域。它是基于Monte-Carlo 迭代求解策略的一种随机寻优算法，其出发点是基于物理中固体物质的退火过程与一般组合优化问题之间的相似性。

模拟退火算法从某一较高初温T出发，T伴随温度参数q的不断下降(T=T\*q),结合一定的概率突跳特性在解空间中随机寻找目标函数的全局最优解，即在局部最优解能概率性地跳出并最终趋于全局最优。

我们在这里先介绍爬山算法。爬山算法是一种简单的贪心搜索算法，该算法每次从当前解的临近解空间中选择一个最优解作为当前解，直到达到一个局部最优解。

爬山算法实现很简单，其主要缺点是会陷入局部最优解，而不一定能搜索到全局最优解。如图1所示：假设C点为当前解，爬山算法搜索到A点这个局部最优解就会停止搜索，因为在A点无论向那个方向小幅度移动都不能得到更优的解。

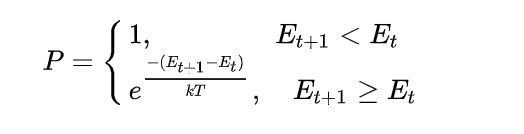


爬山算法

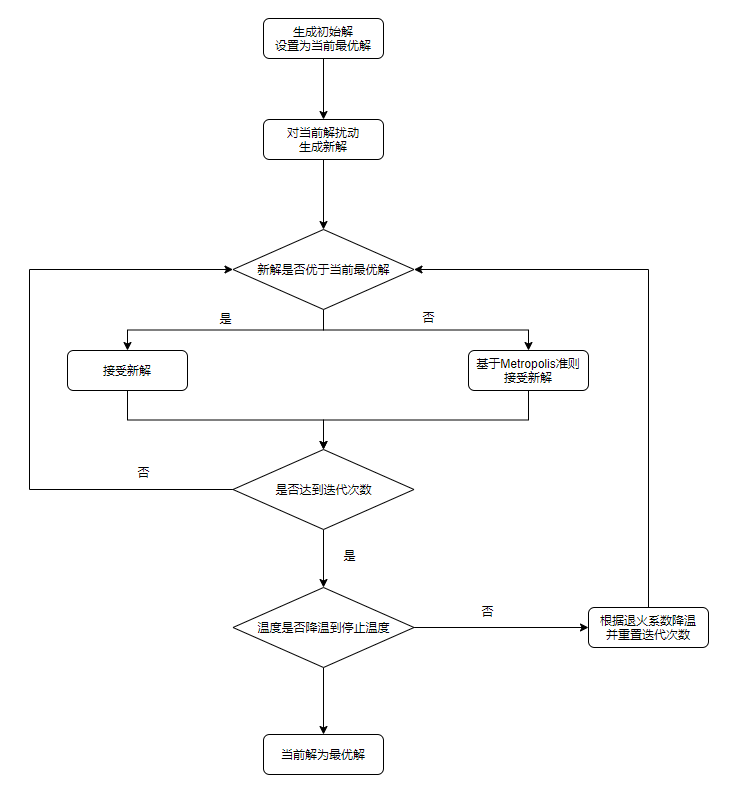
  爬山法是完完全全的贪心法，每次都鼠目寸光的选择一个当前最优解，因此只能搜索到局部的最优值。模拟退火算法**以一定的概率**来接受一个比当前解要差的解，因此**有可能**会跳出这个局部的最优解，达到全局的最优解。以上图为例，模拟退火算法在搜索到局部最优解A后，会**以一定的概率**接受到E的移动。也许经过几次这样的不是局部最优的移动后会到达D点，于是就跳出了局部最大值A。

那么接受解的概率机制是什么呢？如果新解比当前解更优，则接受新解，否则基于Metropolis准则判断是否接受新的次优解。

设为旧解，为新解，T为当前温度，k为常数，则新解的接受概率为：



SA算法实质分两层循环，在任一温度水平下，随机扰动产生新解，并计算目标函数值的变化，决定是否被接受。由于算法初始温度比较高，这样，使E增大的新解在初始时也可能被接受，因而能跳出局部极小值，然后通过缓慢地降低温度，算法就最终可能收敛到全局最优解，具体流程图为：



由此我们可以看出，我们需要根据必经点的多少调节退火系数q以及每个温度的迭代次数L，使退火次数不至于太少而使答案出现在局部最优解，也不至于太多而浪费时间。调节初始温度、终止温度使得降温次数足够，初始跳变概率足够大。

**2.4.2 途径点算法（模拟退火）性能分析**

为计算模拟退火的时间复杂度，我们设T为初始温度，t为终止温度，q为退火系数，a为退火次数，L为迭代次数，可得终止式子：

可算得a：

总时间复杂度为：

与分支界限法优化相比，模拟退火直接优化到了常数等级，堪称降维打击，在a\*L约为**一千万**时，运行时间不超过1s，此时设置超过20个必经点算出来的答案为正确答案。

但随着必经点的增多，**一千万**的次数已经无法算出最优答案，在50个点左右时，答案有小幅波动，当点超过100并跨校区时，会产生往返两次校区的结果。根据退火的原理，我们需要增加更多的退火次数，但设置更多的退火次数后运行时间势必增加，所以我们动态设置了退火次数以及其上限。

此外，随着必经点的增多，单次迭代计算时间也会增多，我们研究发现更多的必经点会造成数据交流的高负荷导致运算时间变高，此问题在Worker中有所改善。

**三、Web Worker**

由于本程序全部基于前端运行，当前端的的数据过多时，就会发现页面产生了明显的卡顿，原因也很简单:

传递数据json需要进行操作，大量计算会阻塞UI线程，导致页面的其他事件无法及时响应，造成页面假死的现象。

那我们能不能把复杂的算法计算操作单独放在一个线程里呢？这时就要请出web worker了。我们使用独立的工作线程运行算法，防止大量计算阻塞UI线程，同时得到了更高的执行效率。

未添加worker的程序退火在**一百万**次左右时运行时间超过1s，而增加worker后**一千万**次运行时间仍小于1s，其优化十分高效。