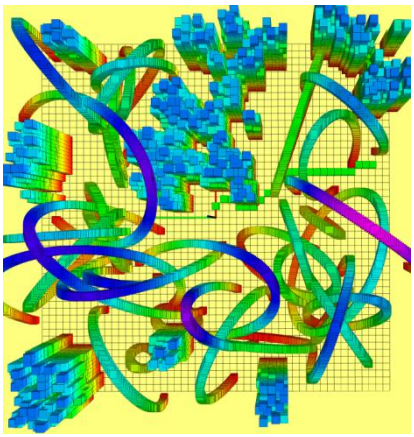
1.A*算法流程如下图所示

```
startNode.g = 0;
startNode.h = heuristicFunc(startNode, goalNode);
startNode.f = startNode.g + startNode.h;
open.insert(startNode);//起点加入open集合中
while(open.empty() == false){
    currentNode = open.begin();//从open中弹出f(n)值最小的节点n;
    if(currentNode == goalNode){//如果到了目标节点
        goalNode.fatherNode = currentNode.fatherNode;//目标节点的父节点替换为当前节点，用于回溯轨迹
        break;
    }
    close.insert(currentNode);//将n加入close队列;
    allNeighborNodes = getNeighborNodes(currentNode);//扩展n节点的邻居neighbors;
    for(neighbor in allNeighborNodes){
        if(neighbor not in open && neighbor not in close){//如果邻居节点是没有访问过的节点
            neighbor.g = currentNode.g + distance(neighbor, currentNode);//更新g值
            neighbor.h = heuristicFunc(neighbor, goalNode);//更新h值
            neighbor.f = neighbor.g + neighbor.h;//更新f值
            neighbor.fatherNode = currentNode;//更新父节点
            open.insert(neighbor);
        }
        else if(neighbor in open){//如果邻居节点是在open集合中的节点
            if(neighbor.g >= currentNode.g + distance(neighbor, currentNode)){//如果邻居节点原本的g大于当前节点的g加上当前节点到邻居节点的距离
                neighbor.g = currentNode.g + distance(neighbor, currentNode);//更新g值
                neighbor.h = heuristicFunc(neighbor, goalNode);//更新h值
                neighbor.f = neighbor.g + neighbor.h;//更新f值
                neighbor.fatherNode = currentNode;//更新父节点
            }
        }
        else if(neighbor in close){
            continue;//对于已经在close集合中的节点不作处理，即跳过;
        }
    }
}
path = backtrack(goalNode);
```

2.A*算法运行效果



```
[ WARN] [1667206917.710253048]: 3D Goal Set
[ INFO] [1667206917.721034701]: [node] receive the planning target
[ WARN] [1667206917.722677686]: [ManhattanHeu A*]{sucess}  Time in A*  is 0.0986
88 ms, path cost if 1.163826 m
[ WARN] [1667206917.723258150]: visited_nodes size : 50
[ WARN] [1667206917.725409592]: [EuclideanHeu A*]{sucess}  Time in A*  is 1.3793
98 ms, path cost if 1.099547 m
[ WARN] [1667206917.726359106]: visited_nodes size : 844
[ WARN] [1667206917.727274119]: DiagnalHeu [A*]{sucess}  Time in A*  is 0.150093
 ms, path cost if 1.140394 m
[ WARN] [1667206917.728217347]: visited_nodes size : 57
```

A*算法的运行效果如上图左边所示，绿色的轨迹为搜索得到的路径，能实现在避障的同时到达目标点。

3.使用不同的启发函数，算法的运行效率如上图右边终端的提示信息所示。

①使用 Manhattan 距离作为启发函数访问总结点数最少，时间最短，因为 Manhattan 函数值相对较大，更加偏向与贪心；②使用 Euclidean 距离作为启发函数访问的总结点数最多，时间最长，因为欧式距离为严格最小的 h 值；③使用对角距离作为启发函数访问的总结点数和运行时间在①②之间，因为在栅格地图下对角距离也是最小值，但比欧式距离大，因此效率比欧式距离高，且得到的同样为最优路径。

①使用 Manhattan 距离作为启发函数得到的路径不一定为最优，因为更偏向于贪心，只考虑当下的收益；②使用 Euclidean 距离作为启发函数得到的路径为最优，因为欧式距离严格最小；③使用对角距离作为启发函数得到的路径为最优，且效率相比与欧氏距离的高，因为栅格地图下对角距离也是最小，但比欧式距离大。**关于不同的启发函数得到的路径优劣程度，在三维的栅格地图下用肉眼判断比较困难，这部分结论根据在 matlab 下二维栅格地图中做的仿真结果得出。**

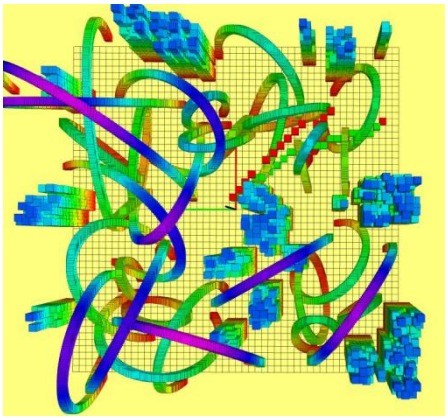
4.加入 Tie Breaker 运行算法的效果如下所示，相对普通的 A*算法，访问的节点数和算法运行时间更短了，Tie Breaker 打破多条相同路径间的平衡性，有偏向性地选择路径，提升了算法的运行效率。

```
[ WARN] [1667218151.866045304]: 3D Goal Set
[ INFO] [1667218151.873664241]: [node] receive the planning target
[ WARN] [1667218151.875891930]: [EuclideanHeu A* without tieBreaker]{sucess}  Ti
me in A*  is 1.114020 ms, path cost if 0.973697 m
[ WARN] [1667218151.877711640]: visited_nodes size : 653
[ WARN] [1667218151.879661942]: [EuclideanHeu A* with tieBreaker]{sucess}  Time
in A*  is 1.064995 ms, path cost if 0.973697 m
[ WARN] [1667218151.880697396]: visited_nodes size : 603
```

5．遇到的问题

①首先是拿到 C++代码后对 ROS 程序的框架不熟悉，根据作业说明文档的提示把核心的算法部分思路理清楚并补完能成功运行后，再从头看 demo_node 的 main 函数和各种回调函数，慢慢了解了框架。②在编写 getHeu 函数时，使用 Manhattan 距离和 Diagnal 距离作为启发函数时，程序运行到一半会崩溃，在 ROS 系统下不方便打断点调试，后来通过 ROS_INFO 输出信息，定位到错误位置，问题是在计算节点 x、y、z 轴坐标之差时没有加绝对值符号，导致距离算出来是负的，程序出错，调整后程序正常运行。

6.JPS 算法



```
[ WARN] [1667221241.837571875]: 3D Goal Set
[ INFO] [1667221241.844091374]: [node] receive the planning target
[ WARN] [1667221241.847724943]: [EuclideanHeu A* without tieBreaker]{sucess}  Ti
me in A*  is 3.514492 ms, path cost if 1.198959 m
[ WARN] [1667221241.848577271]: visited_nodes size : 1063
[ WARN] [1667221241.855886173]: [JPS]{sucess}  Time in JPS  is 6.067146 ms, path
 cost if 4.394793 m
[ WARN] [1667221241.857347856]: visited_nodes size : 3966
```

上图左边是 A*和 JPS 算法在三维栅格地图下的路径搜索结果，绿色的轨迹为 A*算法搜索的路径，红色的轨迹是 JPS 算法搜索的路径，A*算法搜索的路径各个栅格之间是连续的，JPS 算法搜索的路径存在跳跃的情况，是因为这两种算法扩展节点的策略不同。

上图右边是终端中提示的程序运行结果，JPS 算法访问的节点数的运行时间比 A*算法都要多，这是因为随机生成的地图起点周围比较空旷，JPS 需要访问更多的节点，因此搜索的效率比 A*搜索的效率低。而根据课程视频中的内容，在地图比较稠密的情况下，JPS 算法因其特殊的节点扩展策略，搜索的节点数会比 A*算法的少，效率会比 A*算法高。