ROS Notebook

Wu Yutian

2021.11.13

前言

本书的主要内容包括：

- 学习古月居的相关入门课程视频的内容记录
- 阅读胡春旭的《ROS 机器人开发实践》的笔记整理
- 参考高翔的《视觉 SLAM 十四讲》补充了关于三维刚体运动学的内容
- 参考一些博客阅读 ros-navigation 导航包源码的思路整理
-

Wu Yutian
2021.11.13

Contents

Chapter 1

Navigation 详细学习

1.1 global_planner 源码学习

全局路径规划 $BaseGlobalPlanner$ 的 $plugin$ 有三种: $navfn/NavfnROS$,$global\_planner/GlobalPlan$ $carrot\_planner/CarrotPlanner$. 其中常用的为 $global\_planner/GlobalPlanner$,它是 $navfn/NavfnROS$ 的改进版本，包含 $Dijskstra$ 和 $A*$ 算法进行全局路径规划。它与 $Navfn$ 类似，也是提供了一个 $makePlan$ 函数作为它被 $move\_base$ 调用的总接口，负责实现全局的路径规划。

1.1.1 A^* 算法原理

1.1.2 源码相关文件

- 源码链接:

  `https://github.com/ros-planning/navigation/tree/melodic-devel`

- 源码注释链接:

  `https://github.com/W-yt/ROS_Notes/tree/master/navigation-melodic-devel/global_planner`

对应源码中的相关文件:

- global_planner/src/plan_node.cpp
- global_planner/src/planner_core.cpp
- global_planner/src/astar.cpp
- global_planner/src/grid_path.cpp
- global_planner/src/gradient_path.cpp
- global_planner/include/potential_calculator.h

- global_planner/src/quadratic_calculator.cpp
- global_planner/include/expander.h
- global_planner/include/traceback.h

1.1.3 整体结构图

1.1.4 参数配置

参数配置文件为：*global_planner_params.yaml*
参数列表：

- *allow_unknown*：是否允许规划器规划穿过未知区域的路径 (只设计该参数为 *true* 还不行, 还要在 *costmap_commons_params.yaml* 中设置 *track_unknown_space* 参数也为 *true* 才行)
- $default\_tolerance$：当设置的目的地被障碍物占据时, 需要以该参数为半径寻找到最近的点作为新目的地点
- *visualize_potential*：是否显示从 *PointCloud*2 计算得到的势区域
- *use_dijkstra*：设置为 *true*, 将使用 *dijkstra* 算法, 否则使用 A^* 算法
- *use_quadratic*：设置为 *true*, 将使用二次函数近似函数, 否则使用更加简单的计算方式, 这样节省硬件计算资源
- *use_grid_path*：如果设置为 *true*, 则会规划一条沿着网格边界的路径, 偏向于直线穿越网格, 否则将使用梯度下降算法, 路径更为光滑点
- *old_navfn_behavior*：若在某些情况下, 想让 *global_planner* 完全复制 $navfn$ 的功能, 那就设置为 *true*, 但是需要注意 $navfn$ 是非常旧的 *ROS* 系统中使用的, 现在已经都用 *global_planne*r 代替 $navfn$ 了, 所以不建议设置为 *true*
- *lethal_cost*：致命代价值, 默认是设置为 253, 可以动态来配置该参数
- *neutral_cost*：中等代价值, 默认设置是 50, 可以动态配置该参数
- $cost\_factor$：代价地图与每个代价值相乘的因子
- *publish_potential*：是否发布 costmap 的势函数
- *orientation_mode*：如何设置每个点的方向（$None = 0, Forward = 1, Interpolate = 2, ForwardThenInterpolate = 3, Backward = 4, Leftward = 5, Rightward = 6$）（可动态重新配置）
- *orientation_window_size*：根据 *orientation_mode* 指定的位置积分来得到使用窗口的方向. 默认值 1, 可以动态重新配置

1.2 plan_node.cpp

plan_node.cpp 文件中是一个 *PlannerWithCostmap* 类，它继承自 *GlobalPlanner* 类（在 *planner_core* 中定义）。在这个类中，主要开启了两个线程，提供了两种方式去开启 *global_planner* 的规划，一种是服务请求，一种是发布目标 goal 话题。这部分在该类的构造函数中实现，代码如下：

```
PlannerWithCostmap::PlannerWithCostmap(string name, Costmap2DROS* cmap) :
        GlobalPlanner(name, cmap->getCostmap(), cmap->getGlobalFrameID()) {
    ros::NodeHandle private_nh("~");
    cmap_ = cmap;

    //第一个线程提供plan_service
    //一旦有请求，就调用bool success = makePlan(req.start, req.goal, path);
    make_plan_service_ = private_nh.advertiseService("make_plan", &PlannerWithCostmap::makePlanSer
vice, this);

    //第二个线程是去订阅goal
    //拿到goal之后，就调用makePlan(start, *goal, path);
    pose_sub_ = private_nh.subscribe<rm::PoseStamped>("goal", 1, &PlannerWithCostmap::poseCallback,
this);
}
```

1.2.1 planner_core.cpp

planner_core.cpp 文件是 *global_planner* 的核心部分，包括了其中的 *makePlan* 函数，实现了全局路径规划的整体流程，相对与 navfn 的实现，代码更精简、封装更好，十分优雅。

GlobalPlanner::initialize

GlobalPlanner 类的构造函数就是调用 *initialize* 函数，*initialize* 函数中做了参数读取、功能包中用到的类的实例化，并且设置了参数动态配置。

GlobalPlanner::makePlan

该函数中包含了全局路径规划的整体流程逻辑，是该功能包的重点，代码逻辑与 Navfn 类似。

首先将目标点和起始点转换到 *map* 下，然后将起始点设置为 $FREE$，并初始化 *pot* 数组，封闭地图四周，这些部分和 $Navfn$ 相同，只是做了一层封装，用函数调用的方式实现。

接下来调用函数计算 *potential* 值，*planner_* 指针指向 $A*$ 或 *Dijkstrta* 算法类调用其各自的 *calculatePotentials* 函数，这是函数的重点，代码如下：

```
bool found_legal = planner_->calculatePotentials(costmap_->getCharMap(),
                                                  start_x, start_y, goal_x, goal_y,
                                                  nx * ny * 2, potential_array_);
```

接下来，如果在 *calculatePotentials* 中找到了合法的目标点，就可以获取规划的路径了，代码如下：

```
if (found_legal) {
    //根据pot数组获取路径规划结果plan(调用函数getPath 这个函数也有两种实现方式)
    if (getPlanFromPotential(start_x, start_y, goal_x, goal_y, goal, plan)) {
        //确保目标点和其余点有相同的时间戳
        geometry_msgs::PoseStamped goal_copy = goal;
        goal_copy.header.stamp = ros::Time::now();
        plan.push_back(goal_copy);
    } else {
        ROS_ERROR("Failed to get a plan from potential when a legal potential......");
    }
}else{
    ROS_ERROR("Failed to get a plan.");
}
```

然后，给获得的路径数组添加方向信息，并发布规划结果，用于可视化：

```
//添加方向信息(给path “顺毛” 保证拐弯的角度别变得太快)
orientation_filter_->processPath(start, plan);
//发布plan
publishPlan(plan);
```

1.2.2 aster.cpp

该文件中实现了 *aster* 路径搜索算法，实现方式简介优雅（与 *Dijkstra* 算法的复杂实现形成对比），值得借鉴。

需要注意：A^* 算法是策略寻路，不保证一定是最短路径；*Dijkstra* 算法是全局遍历，确保运算结果一定是最短路径，*Dijkstra* 算法且算法需要载入全部数据，遍历搜索。

AStarExpansion::calculatePotentials

从起点开始逐渐扩散，将节点放入 *open* 堆，根据每个 *cell* 的代价值 (*cost* 值) 实现一个小顶堆。实现启发式搜索，不断计算更新经过的节点的 *pot* 值。

附带详细注释的代码如下：

```
bool AStarExpansion::calculatePotentials(unsigned char* costs,
                                         double start_x, double start_y,
                                         double end_x, double end_y,
                                         int cycles, float* potential) {
    //清空队列
```

```
//queue_是启发式搜索到的向量队列<i,cost>
queue_.clear();
//toIndex函数获取索引值
int start_i = toIndex(start_x, start_y);
//将起点放入队列
queue_.push_back(Index(start_i, 0));

//std::fill: 将一个区间的元素都赋予指定的值, 即在 (first, last)范围内填充指定值
//将所有点的potential都设为一个极大值,potential就是估计值g,f=g+h
//potential为g, 也就是从起点到当前点的代价(Dijkstra算法中只有这个值)
std::fill(potential, potential + ns_, POT_HIGH);
//起点的potential值为0
potential[start_i] = 0;

//终点的索引
int goal_i = toIndex(end_x, end_y);
int cycle = 0;

//循环目的: 得到最小cost的索引, 并删除它, 如果索引指向goal则退出算法, 返回true
while (queue_.size() > 0 && cycle < cycles) {
    //将首元素放到最后, 其他元素按照Cost值从小到大排列
    Index top = queue_[0];
    //pop_heap()是将堆顶元素与最后一个元素交换位置, 之后用pop_back将最后一个元素删除
    //greate
    r1()是自己定义的一个针对Index的比较函数, 这里表示依据Index的小顶堆
    std::pop_heap(queue_.begin(), queue_.end(), greater1());
    queue_.pop_back();

    //小顶堆 所以top就是cost最小的点的索引
    int i = top.i;
    //如果到了目标点就结束
    if (i == goal_i)
        return true;

    //对前后左右四个点执行add函数, 将代价最小点i周围点加入搜索队里并更新代价值
    add(costs, potential, potential[i], i + 1, end_x, end_y);
    add(costs, potential, potential[i], i - 1, end_x, end_y);
    add(costs, potential, potential[i], i + nx_, end_x, end_y);
    add(costs, potential, potential[i], i - nx_, end_x, end_y);
```

```
            cycle++;
        }
        return false;
    }
```

AStarExpansion::add

该函数向 *open* 小顶堆中添加节点，并更新代价函数。如果是已经添加的点则忽略，根据 *costmap* 的值如果是障碍物的点也忽略。

更新代价函数的公式：

$$f(n) = g(n) + h(n) \tag{1.1}$$

其中，$g(n)$ 和 $h(n)$ 的意义见注释，代码如下：

```
void AStarExpansion::add(unsigned char* costs,
                         float* potential, float prev_potential,
                         int next_i,
                         int end_x, int end_y) {
    //越界了
    if (next_i < 0 || next_i >= ns_)    return;

    //未搜索的点cost为POT_HIGH，如小于该值，则为已搜索点，跳过
    if (potential[next_i] < POT_HIGH)   return;

    //障碍物或者无信息的点
    if(costs[next_i]>=lethal_cost_ &&
       !(unknown_ && costs[next_i]==costmap_2d::NO_INFORMATION))
        return;

    //potential[next_i]是起点到当前点的cost即g(n)
    //prev_potentia是父节点的pot值
    potential[next_i] = p_calc_->calculatePotential(potential, costs[next_i] + neutral_cost_, next_i,
prev_potential);

    int x = next_i % nx_, y = next_i / nx_;
    //计算distance：从节点n到目标点最佳路径的估计代价，这里选用了曼哈顿距离（不能斜着走 且无视障碍物）
    float distance = abs(end_x - x) + abs(end_y - y);

    //distance只是格子个数，还有乘上每个格子的真实距离或是分辨率，所以最后h = distance*neutral_cost_
    //因此最后的f = h + g = potential[next_i] + distance*neutral_cost_
    //neutral_cost_默认值为50
    queue_.push_back(Index(next_i, potential[next_i] + distance * neutral_cost_));

    //插入小顶堆
    std::push_heap(queue_.begin(), queue_.end(), greater1());
```

```
}
```

1.2.3 grid_path.cpp

因为如 A^* 算法已经完成了路径搜索的话，其实要获取路径只是需要从 *goal* 出发逆向走一遍就好了，所以这一部分要想实现很简单，但是这也只是一种简单的实现方式：创建一条沿着网格边界的路径；也可以采用梯度下降法，使用与 $navfn$ 中相同的梯度下降算法实现方式 (在 *gradient_path.cpp* 中实现)。

getPath 函数实现代码如下：

```
bool GridPath::getPath(float* potential,
                       double start_x, double start_y,
                       double end_x, double end_y,
                       std::vector<std::pair<float, float> >& path) {
    //将goal的坐标放入current中
    std::pair<float, float> current;
    current.first = end_x;
    current.second = end_y;

    int start_index = getIndex(start_x, start_y);

    path.push_back(current);
    int c = 0;
    int ns = xs_ * ys_;

    while (getIndex(current.first, current.second) != start_index) {
        float min_val = 1e10;
        int min_x = 0, min_y = 0;
        //从周围的8个点中寻找pot值最小的点
        for (int xd = -1; xd <= 1; xd++) {
            for (int yd = -1; yd <= 1; yd++) {
                if (xd == 0 && yd == 0)
                    continue;
                int x = current.first + xd, y = current.second + yd;
                int index = getIndex(x, y);
                if (potential[index] < min_val) {
                    min_val = potential[index];
                    min_x = x;
                    min_y = y;
                }
            }
```

```
            }
            if(min_x == 0 && min_y == 0)
                return false;
            current.first = min_x;
            current.second = min_y;
            path.push_back(current);

            //设置了一个循环次数的上限
            if(c++ > ns*4){
                return false;
            }

        }
        return true;
    }
```

1.2.4 gradient_path.cpp

梯度下降算法实现路径获取的实现方式，与 $navfn$ 中的实现相同，不再赘述。

1.2.5 potential_calculator.h

$calculatePotential()$ 计算根据 $use\_quadratic$ 的值有下面两个选择：
该函数计算 pot 值。

- 若为 True 则使用二次曲线计算
- 若为 False 则采用简单方法计算

简单方法即直接返回如下算式：

$$\begin{aligned} newpot &= prev\_potential + cost \\ &= prev\_potential + costs[next\_i] + neutral\_cost\_ \\ &= \text{之前路径代价 g} + \text{地图代价} + \text{单格距离代价 (初始化为 50)} \end{aligned} \tag{1.2}$$

代码如下：

```
    virtual float calculatePotential(float* potential, unsigned char cost, int n, float prev_po
tential=-1){
        //如果父节点的pot值小于0(调用时没有赋值 使用了缺省值-1)
        //(在函数clearEndpoint中会出现这种缺省调用的情况)
        if(prev_potential < 0){
            //则将周围四个点的pot的最小值当做父节点的pot值
```

```
            float min_h = std::min(potential[n - 1], potential[n + 1]);
            float min_v = std::min(potential[n - nx_], potential[n + nx_]);
            prev_potential = std::min(min_h, min_v);
        }

        return prev_potential + cost;
    }
```

其实实现的代码就是直接 *return* 就好了，但是为了适配 *clearEndpoint* 函数也可以直接调用，设置了缺省值，添加了上面的部分。

1.2.6 quadratic_calculator.cpp

该函数计算 pot 值，采用了与 navfn 中相同的二次曲线的计算方法，不再赘述。

1.2.7 expander.h

这个文件中定义了 *Expander* 类，这是两个路径搜索算法（*astar* 和 *dijkstra*）的父类，如果想要自己实现一个路径搜索算法也可以考虑继承该类。具体代码没有什么内容，不再介绍。

1.2.8 traceback.h

这个文件中定义了 *Traceback* 类，这是两个路径获取算法（*grid_path* 和 *gradient_path*）的父类，如果想要自己实现一个路径获取算法也可以考虑继承该类。具体代码没有什么内容，不再介绍。