ROS 下路径规划仿真实验报告

摘要：ROS(Robot Operating System）是一个机器人软件平台，在 ROS 上可以实现对机器人路径、避障等的仿真。在给的库文件中，已经实现了 Dijsktra 和 A*算法，我们在实现两种算法的同时，也自己完成了两种算法，并实现了很好的避障效果。

关键词：路径规划 ROS A* theta*

一、基础算法—A*算法

A*算法是一种启发式搜索算法，通过定义起始点某一节点的实际代价 g(n)和到目标距离的估计代价 h(n)，来寻找最优路径。在 navigation 包中，已经给出了 A*的算法代码，以下是对代码的简要分析。

queue_.clear();是创建并清空栈，以存放路径；

queue_.push_back(Index(start_i, 0));将初始点放入栈；

std::fill(potential, potential + ns_, POT_HIGH);初始化全图的代价估计 g(n)，同时用来检测节点是否被搜索；

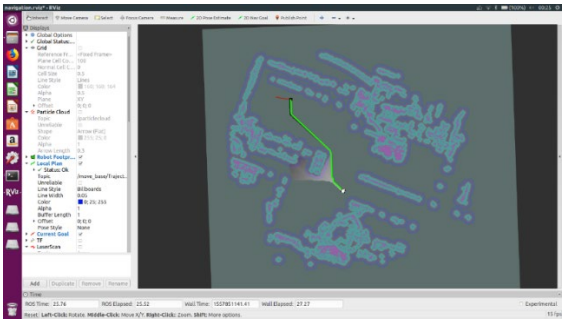
std::pop_heap(queue_.begin(), queue_.end(), greater1());

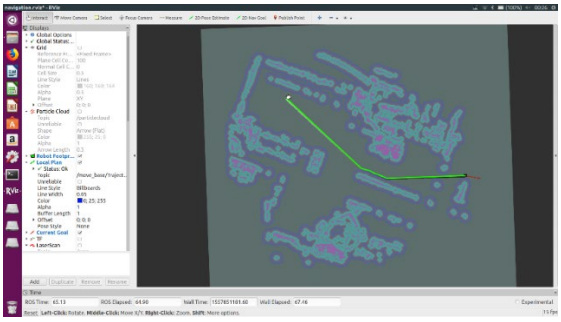
queue_.pop_back();是将下一个目标点排序后取最小值放入栈中；

add(costs, potential, potential[i], i + 1, end_x, end_y);第四个形参分别为当前点的上下左右点，将不是障碍物的点放入栈中，同时标记这几个点的 potential 不等于 POT_HIGH 以示搜索过该点。

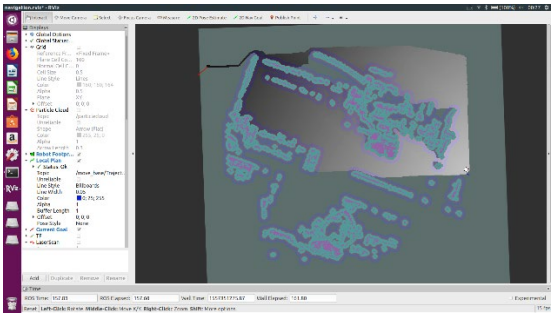
其中，potential[next_i] = p_calc_->calculatePotential(potential, costs[next_i] + neutral_cost_, next_i, prev_potential);代码是计算某一点的 potential，在 A*.h 中文件定义。

以下是实验中对 A*算法实际效果的截图





可见，A*算法搜索范围较小，并且可以找到相对代价小的路径，但是，当起点与目标点相距较远时，容易出现无法找到路径的情况：



可见，A*在搜索了很大的区域后，并没有给出对应的路径，因此 A*在应用上有一定的局限性。

二、一种简单的避障算法实现路径规划

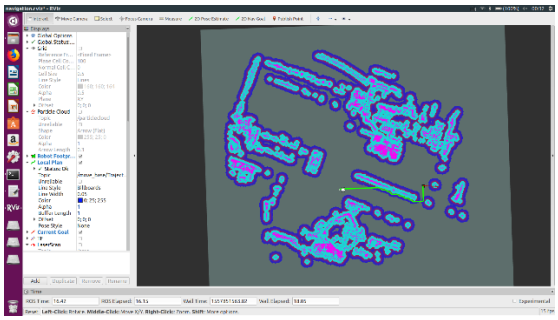
运用的 bug 避障算法的一些思想，首先机器人在 x 坐标轴上到达与目标点相同的 x 坐标处，之后再在 y 轴上到达目标点。以下是对代码的分析。

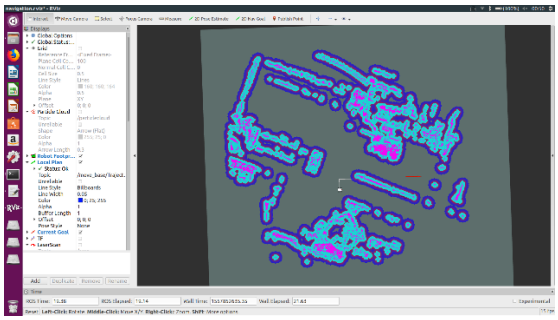
我们运用了 A*与 global_planner 包的接口，即通过 push_back 和 push_heap 命令来实现入栈出栈的操作来提供目标点的坐标。

（图，可行的是发过去的那张图，就是那个障碍物的右上方且下面没有障碍物）

优点，在 x 轴上可以很好的实现避障功能，并且计算量很小，不需要对环境的提前感知，只需要在行进过程中进行障碍物检测。

缺点，由于算法的优化问题，目前仅仅能实现 x 轴上还未到达目标 x 坐标时的避障，由于避障需要使用目标点与当前位置的 x 轴坐标差,因此,当这个数字变为零之后，在 y 轴上就失去了避障的能力。下图分别为为路径规划成功和失败时的实际效果：





以下是对代码的简单解释：

首先我定义了一个 direct_1 变量，来表示是否已经到达与目标点相同的 x 坐标处，若等于 1，则表示需要沿 x 轴前行，若等于 2，则表示需要沿 y 轴前行:next_i=i+direct_x，表示沿 x 轴前行；next_i=i+direct_y*nx_表示沿 y 轴前行。

避障算法：当检测到下一个节点 next_i 为障碍物时，则进入避障循环：

```
if (next_i < 0 || next_i >= ns_ ||
potential[next_i] <
POT_HIGH||(costs[next_i]>=lethal_cost
_-1 && !(unknown_ &&
costs[next_i]==costmap_2d::NO_INFORMA
TION))){
        next_i=i-cor_x-cor_y*nx_;
        if(direct_3<=5)
            direct_1++;
        direct_3=0;
    }
```

首先机器人向与目标点相反的 x、y 方向移动一个像素点（若此时 x=0，则会出现在 y 方向循环而无法避障的情况），之后重新进行之前的导航，直到到达目的地。

其中 direct_3 变量的目的是检测避障前走过的格数，以帮助实现 y 轴上的避障，但很遗憾还没有成功。

三、theta*算法

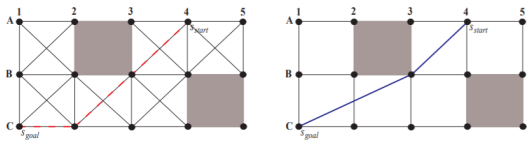
1、估值函数优化：

传统的Astar算法所用的曼哈顿距离往往会遇到两边代价相同的情况，造成更多的计算。这里我们用一个新的估值函数来避免这个问题的产生。

```
int estimateValue(Point gaol,Point
current)
{
    int dx = abs(goal.x-current.x);
    int dy = abs(goal.y-current.y);
    if(dx > dy)
        return 10*dx+4*dy;
    else
        return 10*dy+4*dx;
}
```

2、Theta*算法

传统的 A*算法是按照一个格子一个格子的顺序进行路径规划的，导致了一个问题：物体只能通过 8 个方向行动，而且必须像跳棋一样一个格子一个格子前进。在这里我们可以通过 theta*算法来对 A*算法进行改进。



(a) Grid path (b) True shortest path

左侧为 A*算法实现得到的路径，右侧为我们思考看出来的最优路径，上图可以直观地表现出 A*算法的缺陷。

Theta *和 A *之间的关键区别在于 Theta *允许顶点的父节点是任何顶点，不像 A *，其中父节点必须是可见邻节点。theta*算法是A*的一种改进，这类算法比A*多了一个收缩父节点的操作，关键在于其打开一个节点 s，然后更新周围的节点 s’时，会检查 s’与 parent（s）的可见性。如果可见，则把 s’的父节点设置成 parent(s)。

伪代码：

```
1  Main()
2      open := closed := ∅;
3      g(s_start) := 0;
4      parent(s_start) := s_start;
5      open.Insert(s_start, g(s_start) + h(s_start));
6      while open ≠ ∅ do
7          s := open.Pop();
8          if s = s_goal then
9              return "path found";
10         closed := closed ∪ {s};
11         foreach s' ∈ nghbr_vis(s) do
12             if s' ∉ closed then
13                 if s' ∉ open then
14                     g(s') := ∞;
15                     parent(s') := NULL;
16                 UpdateVertex(s, s');
17     return "no path found";
18 end
19 UpdateVertex(s, s')
20     g_old := g(s');
21     ComputeCost(s, s');
22     if g(s') < g_old then
23         if s' ∈ open then
24             open.Remove(s');
25         open.Insert(s', g(s') + h(s'));
26 end
27 ComputeCost(s, s')
28     if lineofsight(parent(s), s') then
29         /* Path 2 */
30         if g(parent(s)) + c(parent(s), s') < g(s')
           then
31             parent(s') := parent(s);
32             g(s') := g(parent(s))+c(parent(s), s');
33     else
34         /* Path 1 */
35         if g(s) + c(s, s') < g(s') then
36             parent(s') := s;
37             g(s') := g(s) + c(s, s');
38 end
```

从伪代码的标红中可以看出，theta*算法比 A*算法只多了视线检测和父节点更换操作。

视线检测函数：LineOfSight

只需对方格网格进行整数运算，就可以非常有效地执行视线检查。这样做的原因是执行视线检查类似于在两点之间绘制直线时确定在光栅显示上绘制哪些点。绘制的点对应于直线穿过的单元格。因此，当且仅当没有绘制的点对应于被阻挡的单元格时，两个顶点具有视线。

伪代码：

```
LineOfSight(s, s')
    x_0 := s.x;
    y_0 := s.y;
    x_1 := s'.x;
    y_1 := s'.y;
    d_y := y_1 − y_0;
    d_x := x_1 − x_0;
    f := 0;
    if d_y < 0 then
        d_y := −d_y;
        s_y := −1;
    else
        s_y := 1;
    if d_x < 0 then
        d_x := −d_x;
        s_x := −1;
    else
        s_x := 1;
    if d_x ≥ d_y then
        while x_0 ≠ x_1 do
            f := f + d_y;
            if f ≥ d_x then
                if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                    return false;
                y_0 := y_0 + s_y;
                f := f − d_x;
            if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                return false;
            if d_y = 0 AND grid[x_0 + ((s_x − 1)/2), y_0] AND grid[x_0 + ((s_x − 1)/2), y_0 − 1] then
                return false;
            x_0 := x_0 + s_x;
    else
        while y_0 ≠ y_1 do
            f := f + d_x;
            if f ≥ d_y then
                if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                    return false;
                x_0 := x_0 + s_x;
                f := f − d_y;
            if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
                return false;
            if d_x = 0 AND grid[x_0, y_0 + ((s_y − 1)/2)] AND grid[x_0 − 1, y_0 + ((s_y − 1)/2)] then
                return false;
            y_0 := y_0 + s_y;
    return true;
end
```

虽然 theta*算法可以大大缩短路径距离，然而，theta*有一个很大的问题，就是需要做大量的 LineOfSight 检查。有多少个点进入过 open 列表，就有多少次检查。在较为细致的网格中这个数量是十分巨大的。这样就产生了新的改进算法 lazytheta*算法。

3、lazytheta*算法

在 Theta*中，检查视线的时机发生于一个点进入 open 列表的时候。但是实际上，有很多进入open列表的点最终不在路径中，这意味着视线检查是无效的。因此 Lazy

Theta*选择把视线检查放到打开该节点的时候进行。进入 open 列表的点，我们需要设置它的 g 值和 parent。在 Lazy Theta*中，我们乐观地认为在这里视线检查永远成立的，因此 g 值和 parent 值得的设置按照 Theta*中视线检查成立一般地进行设置。随后，在打开一个节点时，我们对这个点调用 SetVertex 方法。该方法中我们对该点和它的父节点进行真正的视线检查。如果成立，那么我们之前的假设是对的，那就继续进行下去。如果没有视线，那么我们还需要为这个点找到一个正确的 parent。我们直接取 s 的相邻点作为 parent 的候选。然后和 close 表做个交集，在其中选择最好的点即可。

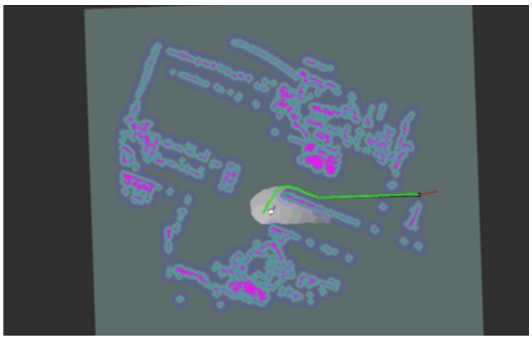
伪代码：

```
1  Main()
2      open := closed := ∅;
3      g(s_start) := 0;
4      parent(s_start) := s_start;
5      open.Insert(s_start, g(s_start) + h(s_start));
6      while open ≠ ∅ do
7          s := open.Pop();
8          SetVertex(s);
9          if s = s_goal then
10             return "path found";
11         closed := closed ∪ {s};
12         foreach s' ∈ nghbr_vis(s) do
13             if s' ∉ closed then
14                 if s' ∉ open then
15                     g(s') := ∞;
16                     parent(s') := NULL;
17                 UpdateVertex(s, s');
18     return "no path found";
19 end
20 UpdateVertex(s, s')
21     g_old := g(s');
22     ComputeCost(s, s');
23     if g(s') < g_old then
24         if s' ∈ open then
25             open.Remove(s');
26         open.Insert(s', g(s') + h(s'));
27 end
28 ComputeCost(s, s')
29     /* Path 2 */
30     if g(parent(s)) + c(parent(s), s') < g(s') then
31         parent(s') := parent(s);
32         g(s') := g(parent(s)) + c(parent(s), s');
33 end
34 SetVertex(s)
35     if NOT lineofsight(parent(s), s) then
36         /* Path 1*/
37         parent(s) :=
           argmin_{s'∈nghbr_vis(s)∩closed}(g(s') + c(s', s));
38         g(s) :=
           min_{s'∈nghbr_vis(s)∩closed}(g(s') + c(s', s));
39 end
```

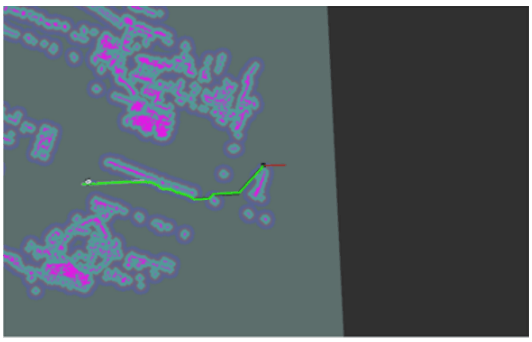
四、A*算法与 theta 算法的对比

对于同一个路线，我们对比了 A*和 theta*的区别如下图：

第一步：

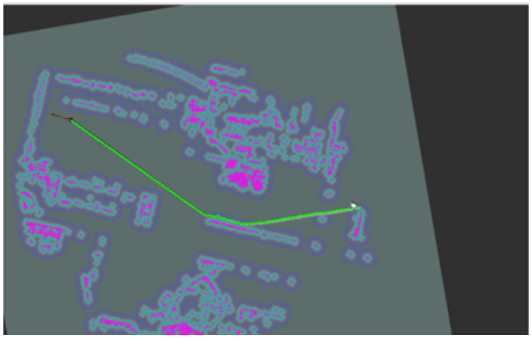


A*

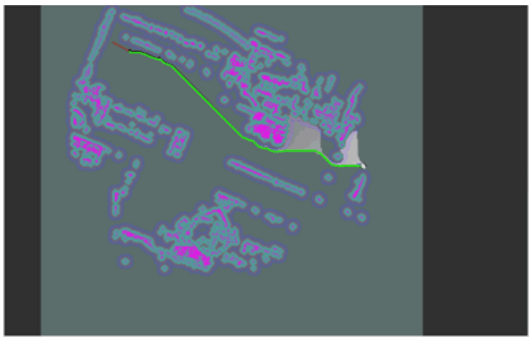


Theta*

第二步：

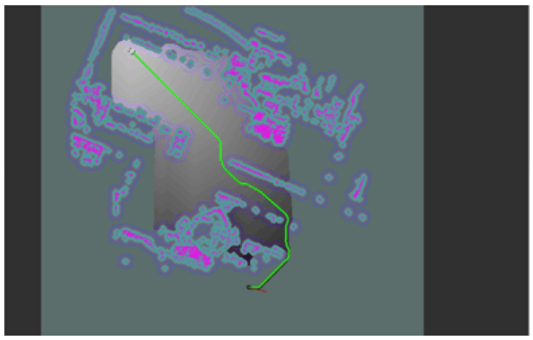


A*

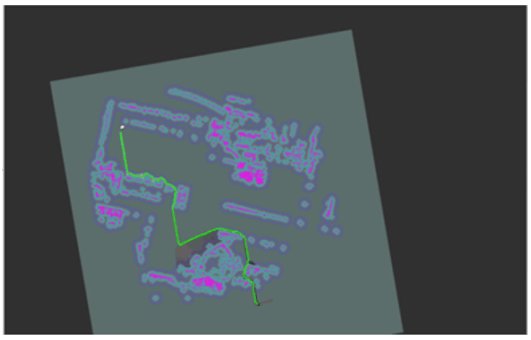


Theta*

第三步：

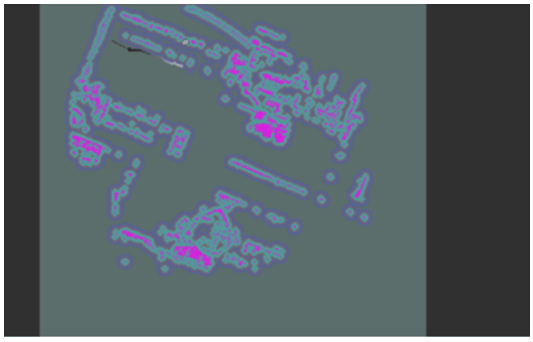


A*



Theta*

同时，在第三步时，A*发生了如下错误，无法导航直到我们将其重新放置。而theta*完全未出现错误



通过对比可以发现，theta*面对较为复杂的路径规划效果远超于 A*,而且不会出现找不到路的情况,并且在时间上由于lazy算法的提升也不逊色于 A*。

五、总结展望：

通过 theta*对 A*算法进行优化，我们可以看到其效果在一定程度上优于 A*,实验成功。但仍有不足之处，可供读者参考，有待优化：

1、双向搜索：可进一步使用双向搜索增加效率

2、递归 theta：在每一步再用 theta*，使其平滑化

3、优化估值函数，使其计算速度和效率最优

参考文献：

[1]A. Nash and S. Koenig and C. Tovey, Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D, (2010) Proceedings of the AAAI Conference on Artificial Intelligence

[2]A. Nash, K. Daniel, S. Koenig and A. Felner , Theta*: Any-Angle Path Planning on Grids, (2007) Proceedings of the AAAI Conference on Artificial Intelligence