

# Taller 12: Planificación de caminos utilizando A\*

Introducción a la Robótica Móvil

May 30, 2018

## 1 Introducción

El problema de la planificación de caminos (o trayectorias) es de gran importancia en la robótica móvil y existen muchas formas de abordarlo. Dependiendo de la representación del mapa que dispongamos y las limitaciones de movimiento de la plataforma robot, se han ideado diversos algoritmos para resolverlo. En este taller trabajaremos con el algoritmo de planificación llamado **A\*** y utilizaremos una **grilla de ocupación** 2D como mapa.

## 2 Estructura del código

Para la realización de este taller se trabajará en el paquete **astar\_planning** y deberán completar la implementación del archivo `/src/AStarPlanner.cpp`, específicamente el método:

```
bool do_planning(Trajectory& result_trajectory)
```

Este deberá devolver **true** en caso de haber encontrado una trayectoria válida y volcar dicha trayectoria en la referencia **result\_trajectory**.

La planificación de la trayectoria se debe realizar en base a la información provista de las siguientes **variables globales**:

- **grid\_** : Mensaje de **OccupancyGrid** recibido.
- **goal\_pose\_** : Posición del goal en referencia al marco de coordenadas del mapa.
- **starting\_pose\_** : Posición del inicial del robot en referencia al marco de coordenadas del mapa.

**NOTA:** Leer con atención los comentarios del código para entender como acceder y utilizar la información provista.

### 2.1 Métodos auxiliares para el manejo de la grilla

Las grillas de ocupación tienen la particularidad de poseer su **propio origen de coordenadas** y no es trivial determinar a que celda hacen referencia las posiciones **x,y** del entorno.

Para facilitar esta tarea se proveen las siguientes funciones auxiliares:

```
bool getOriginOfCell(uint i, uint j, double& x, double& y);
bool getCenterOfCell(uint i, uint j, double& x, double& y);
```

Permiten obtener las posiciones **x,y** del centro y origen (esquina) de una celda **i,j**. Retorna false si la celda provista es inválida y las coordenadas **x,y** **por referencia**.

```
bool getCellOfPosition(const double& x, const double& y, uint& i, uint& j);
```

Permite obtener la celda **i,j** que le corresponde a la posición **x,y**. Retorna false si la posición provista **esta por fuera** de la superficie abarcada por el mapa y la celda **i,j** **por referencia**.

```
bool isCellOccupied(uint i, uint j);
bool isANeighborCellOccupied(uint i, uint j);
bool isPositionOccupied(const double& x, const double& y);
```

Permiten comprobar si una celda, vecindad de celdas o una posición se encuentra ocupada. Retornan por defecto false si las celdas son inválidas o la posición esta por fuera de la superficie cubierta.

## 2.2 Estructuras auxiliares

Para la implementación del algoritmo **A\*** se proveen dos estructuras auxiliares:

```
struct Cell
{
    uint i;
    uint j;

    Cell(uint _i, uint _j);
}

struct CellWithPriority : Cell
{
    double priority;

    CellWithPriority(uint _i, uint _j,
                    const double& _priority);
}
```

Se recomiendan utilizar estas estructuras dado que permiten la utilización de las **colas de prioridad** de la librería estandar de C++:

```
CellWithPriority celda1(0,0 , 5);
CellWithPriority celda2(3,2 , 0);

std::priority_queue<CellWithPriority> cola_min;

cola_min.push(celda1);
cola_min.push(celda2);

CellWithPriority celda_menor_costo = cola_min.top();
cola_min.pop(); // Descarta el tope de la cola
```

En el ejemplo **celda\_menor\_costo** resulta asignada con **celda2**.

## 2.3 Representación del camino y la trayectoria

**A\*** debe ir construyendo el camino a medida que explora la grilla de ocupación. Para esto se recomienda utilizar diccionarios de la librería estándar de C++ de manera de representar el grafo de relaciones:

```
std::map<Cell, Cell> came_from;

// Ejemplo de camino
Cell goal(2,0);
Cell medio(1,0);
Cell start(1,0);

// Grafo: goal -> medio -> start
came_from[goal] = medio;
came_from[medio] = start;

notifyTrajectory(result_trajectory, start, goal, came_from);
```

**notifyTrajectory** recorre el grafo del camino construido por las relaciones establecidas en el **diccionario** **came\_from** y construye los **waypoints**  $(x, y, \theta)$  correspondientes estableciendo el ángulo  $\theta$  en cada momento como el ángulo existente entre los puntos del camino encontrado.

## Ejercicio: Planificación de caminos con **A\*** y seguimiento de trayectorias con el Pioneer

Para este ejercicio se buscará enlazar la planificación de caminos con los métodos de control y seguimiento de trayectorias vistos con anterioridad en la materia.

Para esto deberán trabajar con la escena **V-Rep**:  
`/astar_planning/vrep/astar_planning.ttt`

Y la configuración de **RViz**:  
`/astar_planning/launch/astar_planning.rviz`

Para ejecutar los nodos necesarios utilizar:

```
roslaunch astar_planning astar_planning.launch
```

- Implementar el algoritmo de **A\*** utilizando la heurística de costo Manhattan. ¿El robot llega a destino sin chocarse? ¿Cual es el problema?
- Implementar la búsqueda de la vecindad de una celda excluyendo aquellas vecinas que "linden" con una ocupada.

Se recomienda considerar la operación auxiliar `isANeighborCellOccupied(i,j)` y completar el método:

```
std::vector<Cell> neighbors(const Cell& c)
```

**NOTA:** Pueden modificar la locación de los obstáculos y el goal siempre y cuando se mantengan **completamente contenidos en una celda del piso** (o aproximadamente).

- ¿El robot siempre se mantiene sobre el camino planificado? ¿Por que y como se podría mejorar?