RRT+ Algorithm

Luis Menéndez García
Msc Robotics and Automation
ragnemul@gmail.com

***Abstract*—Se describe el diseño e implementación de un algortimo basado en Rapidly Exploring Random Trees, en adelante denominado RRT+, como un subtipo de Rapidly Exploring Dense Trees con el empleo de técnicas de muestreo aleatorias. Se aportan las posibles modificaciones que pueden ser implementadas como continuación del trabajo presentado.**

Keywords—Tress, RRT+, RRT, RDT.

I. INTRODUCCIÓN

Se ha de dar solución a un problema de planificación sobre plano para un robot móvil que puede moverse sobre el espacio de configuraciones $\mathcal{R}^2 x S_1 = SE(2)$, para que desplace entre dos puntos cualesquiera del mapa, con la premisa de que se conoce el plano de antemano, y éste es estático. Para conseguir estos objetivos se ha desarrollado una variante del algoritmo RRT descrita en [1], denominada RRT+, que permite realizar un camino de conexión libre de obstáculos entre el punto inicial q_i y el final q_f, sin necesidad de completar el muestreo aleatorio de todos los puntos del espacio de configuraciones libre C_{free}, dando como resultado el camino de conexión más corto entre q_i y q_f, según el grafo de conectividad $\mathcal{G}$ que se haya ido formando, o bien mostrando que no hay camino posible debido a una insuficiencia del muestreo de puntos aleatorios sobre el espacio C_{free}.

II. DESCRIPCIÓN GENERAL DEL ALGORITMO

El algoritmo necesita como entrada un conjunto de muestreo aleatorio sobre C_{free}, distribuido uniformemente, y que se denominara C_{sam}, sobre el cual se construirá el grafo $\mathcal{G}$ de conectividad empleando una métrica L^2, así como los puntos de inicio y final q_i y q_f respectivamente, y el mapa binario que representa tanto los espacios pertenecientes a C_{free}, representados con el número 0, así como el conjunto de configuraciones no disponibles C_{obs} representados con el número 1.

RRT+ $(q_i, q_f, C_{sam}, C_{obs}, \mathcal{G})$
1. $\mathcal{G}.init(q_i)$
2. ***while*** $!empty(C_{sam})$
3. $p = GetFirst(C_{sam})$
4. n = $Nearest(\mathcal{G}, p, C_{obs})$
5. **if** $(!n)$
6. $C_{sam} = InsertAtEnd(C_{sam}, p)$
7. **continue**
8. **if** $FreePath(\mathcal{G}, C_{obs}, q_f)$
9. **break**
10. $\mathcal{G}.add\_edge(n, p)$
11. $\mathcal{G}.add\_edge(n, q_f)$

Como puede observarse el algoritmo comienza introduciendo el punto de salida q_i, como un vértice sobre el grafo $\mathcal{G}$ (línea 1), para continuar explorando el conjunto de muestreo aleatorio C_{sam}, que en esta variación sobre el algoritmo original se trata como una pila. Se extrae el último punto p de C_{sam} (línea 3) y se obtiene el vértice n del grafo conectado $\mathcal{G}$ más cercano al punto p que puede formar un camino libre de obstáculos (línea 4). Si no se puede obtener un path libre entre los puntos n y p, se vuelve a encolar el punto p al final de C_{sam} y se fuerza a explorar una nueva iteración (líneas 5 – 7). Después se comprueba si se puede conectar el punto final q_f con el grafo conectado $\mathcal{G}$, y si es así se interrumpe la ejecución del bucle (líneas 8 y 9). Finalmente se conecta el punto q_f al grafo conectado $\mathcal{G}$ (línea 1). Finalmente se hallará el camino más corto entre los puntos q_i y q_f del grafo conectado $\mathcal{G}$, empleando técnicas de recorridos sobre grafos. En función de la distribución de los puntos de muestreo aleatorio C_{sam}, esta técnica es sensiblemente más rápida que la mostrada por el algoritmo RRT estándar, al no ser preciso construir completamente el grafo $\mathcal{G}$ usando todos los puntos de C_{sam}.

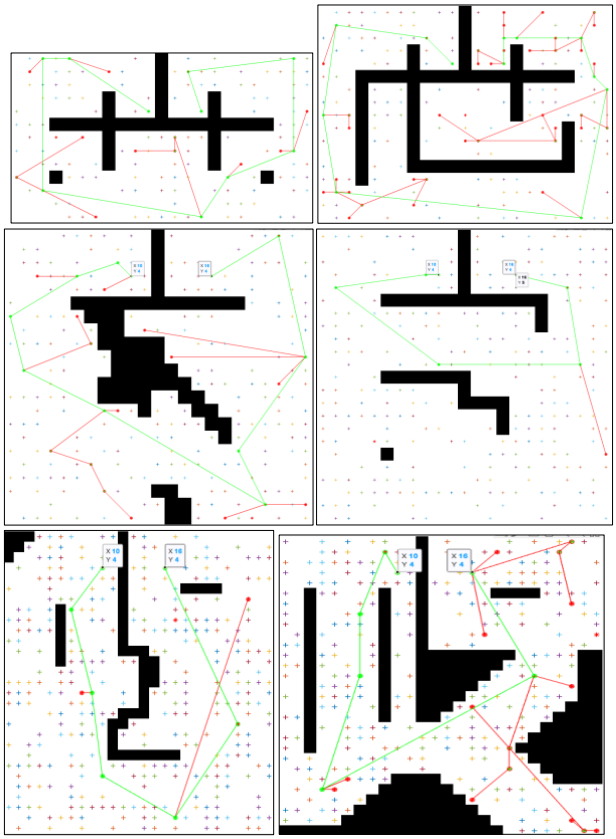

Ilustración 1 Caminos encontrados con exploración parcial

III. MUESTREO ALEATORIO

IV. FUNCIÓN NEAREST

La función NEAREST devuelve el punto $n \epsilon \mathcal{G}$ más cercano al punto explorado $p \epsilon C_{sam}$, siempre que el segmento que une n y p no colisione con ningún obstáculo.

Para la elección del punto $n \epsilon \mathcal{G}$ más cercano se ha empleado la distancia euclídea.

V. Detección de obstáculos

Para la detección de los obstáculos se ha optado por localizar sobre el bitmap los puntos contiguos con conectividad 8.

```
cc = bwconncomp(borderless_map);

  struct with fields:

    Connectivity: 8
       ImageSize: [22 23]
      NumObjects: 2
    PixelIdxList: {[56×1 double]  [6×1 double]}
```

Esta será la estructura base que se consultará cuando se necesite comprobar si el segmento que une $p \in C_{sam}$ con el punto más cercano $n \in \mathcal{G}$, empleando una métrica L^2.

VI. Detección de Path Libre

Se han diseñado e implementado dos algoritmos para la detección del path libre entre dos puntos, que se detallan a continuación.

A. *Primera aproximación para la detección de path libre*

El primer algoritmo –ArcoLibre– se basa en la obtención del perímetro de cada pixel que compone el obstáculo detectado $Pixel_{obs}$, entiendo como pixel la unidad de construcción básica de dicho obstáculo.

En el proceso de construcción del grafo $\mathcal{G}$, para cada segmento candidato a ser incorporado se examina si se produce la intersección con cada uno de los perímetros del conjunto $Pixel_{obs}$. Si no se produce la intersección del segmento éste pasa a considerarse un camino libre y se incorpora a $\mathcal{G}$. En caso contrario el segmento en cuestión se descarta. Este primer algoritmo nos permitirá abordar una evolución del presente trabajo para poder incorporar caminos semilibres, al obtener, si el segmento no es libre, el punto de intersección con C_{obs}.

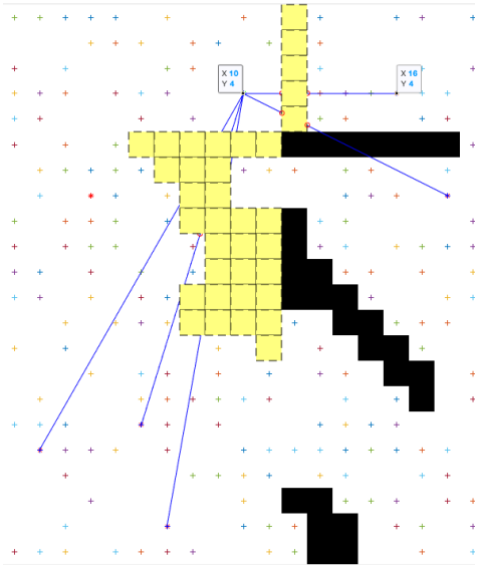

Ilustración 2 Verificación path libre en función ArcoLibre

B. *Segunda aproximación para la detección de path libre*

VII. Obtención del nodo más cercano de $\mathcal{G}$ a $p \in C_{sam}$

VIII. Mejoras a realizar

A. *Cómo evitar los pasos estrechos*

Para evitar que el grafo se quede sin conexión entre los puntos q_i y q_f, al margen de poder utilizar un espacio de muestreo mucho más amplio con el consiguiente incremento del coste computacional y del tiempo necesario para obtener una ruta libre de obstáculos, se propone investigar las características del mapa para averiguar dónde se encuentran los pasos estrechos y así introducir un punto de muestreo justo sobre el.

Esto podría hacerse aplicando una máscara sobre el bitmap con los patrones de acceso que se considera corresponden a pasos estrechos, para posteriormente capturar dichas posiciones y añadir al conjunto C_{sam} los puntos identificados, mostrados en color blanco sobre la imagen de la derecha.

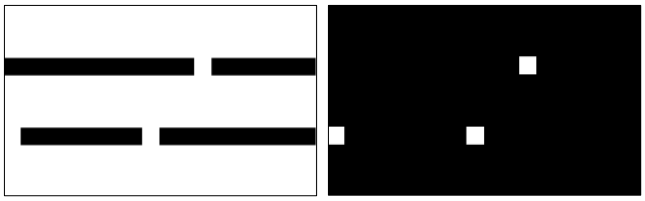

Ilustración 3 Mapa con puntos estrechos y mascara identificando los pasos estrechos

Las siguientes instrucciones de MatLab se han utilizado para obtener los puntos de muestreo de los pasos estrechos mostrados en la ilustración anterior.

```
B1 = [0 0 0;1 0 1;0 0 0];
B2 = [1 1 1;0 1 0;1 0 0];
S1 = bwhitmiss(map,B1,B2);
```

References

[1] LaValle, S. (2006). *Planning Algorithms*. Cambridge: Cambridge University Press. doi:10.1017/CBO9780511546877