PRÁCTICA 1

PRANIFICACIÓN DE CAMINOS EN ROBÓTICA

12 DE ABRIL DE 2018

ernestomar1997@correo.ugr.es

Descripción del Problema

Primero he implementado un algoritmo A^* estándar para resolver el primer apartado intentando tener la mayor eficiencia posible. Posteriormente para el segundo apartado he usado la técnica bidireccional que consiste en trazar dos planes de forma simultanea de destino a origen y viceversa. También he intentado dinamizar los parámetros asociados al coste y a la heurística sacando algunas conclusiones interesantes. Por último, he ajustado el $cost\_map$ para que el robot evalúe la seguridad de los parámetros. Aportaré algunos experimentos positivos y negativos sobre los ejemplos anteriormente mencionados usando medidas de tiempo y optimalidad.

Tarea 1: Implementación del algoritmo A^*

A continuación mostraré el $pseudocodico$ sobre el que he implementado dicho algoritmo. En la columna de la derecha de la siguiente página está indicada la eficiencia conseguida sobre una variable relacionada con el cardinal del conjunto en cuestión.

He utilizado 2 listas que mantengo ordenadas para los nodos abiertos y cerrados:

$$list < node > openList$$

$$list < node > closeList$$

El objetivo de mantenerlas ordenadas es balancear la inserción y la búsqueda sobre ella a una eficiencia logarítmica. Las equivalencias de cpp para este concepto son:

$$upper\_bound(list, value)$$

$$binary\_search(list, value)$$

$$list.insert(position, value)$$

Las heurísticas más conocidas, $euclides$ y $manhattan$, son monótonas por lo que no es necesario la actualización de la lista de cerrados. En este caso como el coste implica el uso de $euclides$ he mantenido la aportada.

También se he alterado la función $footprint$ para que las celdas que choquen con el robot tengan un coste muy alto de tal forma que nos las considere en el algoritmo. Por lo tanto, se mejora la seguridad de los caminos generados.

Nota

Si fuera un caso real con un computador más preparado usaría una estructura de datos en árbol tanto para la lista de cerrados y abiertos, pero con los números que trabajamos en la práctica no mejora los tiempos.

También existen segmentes del código susceptibles de ser paralelizadas, sobre todo las operaciones algebraicas y las búsquedas.

```
a_star(start, goal) {
        openList := {start}
        closeList := ∅
        while (#openList ! = 0) {
                bestnode := openList_1                                  O(1)
                openList := openList − {bestnode}                       O(log(n))
                closedList := binary_insert(bestnode)                   O(log(n))
                if (bestnode == goal) {
                        return (build_way(closedList))                  O(n)
                }
                neighbor := findNeighbor(bestnode)                      O(k)
                inOpenList := (neighbor − closedList) ∩ openList        O(log(n)^2)
                outOpenList := (neighbor − closedList) − openList       O(log(n)^2)
                for (n ∈ outOpenList) {                                 O(nlog(n))
                        n.parent := bestnode
                        n.g := bestnode.g + distance(bestnode, n)
                        n.h := distance(n, goal)
                        n.f := n.g + n.h
                        openList := binary_insert(n)                    O(log(n))
                }
                for (n ∈ inOpenList) {                                  O(nlog(n))
                        m := find(n ∈ openList)                         O(log(n))
                        ten_cost := bestnode.g + distance(bestnode, m)
                        if(ten_cost < m.g)
                                m.parent := bestnode
                                m.g := ten_cost
                        }
                }
        }
        return (ERROR)
}
```

Tarea 2: Mejoras del algoritmo A^*

Agregación de pesos

El algoritmo A^* usa la siguiente función de evaluación:

$$f(n) = g(n) + h(n)$$

Podemos hacer que se tenga más en cuenta el coste que la heurística dependiendo de lo que nos convenga más en cada mapa.

$$f(n) = \alpha g(n) + \beta h(n)$$

El problema es que cada mapa es diferente y en base a los experimentos realizados puedo constatar 2 cosas:

- En mapas semejantes a una distribución en cuadricula el darle más valor a la heurística de media da mejores resultados en la cantidad de nodos abiertos debido a las rectas que debe recorrer.
- Es imposible fijar un valor de α y β óptimo para todos los mapas.

La conclusión que saco es que la asignación de pesos constante no da resultado.

Dentro de los espacios de búsqueda existen dos propiedades fundamentales a la hora de implementar algoritmos:

- Intensificación: propiedad que surge al incrementar la velocidad a la que se llega a un óptimo local.
- Diversificación: propiedad de explorar nuevas zonas del espacio de búsqueda que no han sido previamente exploradas.

He asociado la intensificación con minimizar la $h(n)$, la distancia hasta el objetivo, y la diversificación con expandir nodos que no tienen una $h(n)$ tan prometedora.

El problema es establecer medidores que reflejen de forma dinámica las anteriores propiedades.

- Refleja la proporción de nodos abiertos y cerrados. La clave de la relación es pensar que cuando se encuentra con un obstáculo la cantidad de nodos cerrados aumenta de forma exponencial y el índice propuesto disminuye. Mediante experimentos lo he constado y he detectado un error en el planteamiento. Una vez que el obstáculo es superado reduciendo la consideración de la heurística dicho parámetro no se relaja. Por lo cual planteo el siguiente modelo.

$$\frac{\#openList}{\#closeList}$$

- Refleja la variación en crecimiento o decrecimiento de la proporción en un numero de iteraciones k. Sería la equivalencia física a la velocidad y en nuestro tema la complejidad que presenta el camino. Este índice dinámico si es capaz de relajar y acelerar la intensificación.

$$\frac{\frac{\#openList_n}{\#closeList_n}}{\frac{\#openList_{n+k}}{\#closeList_{n+k}}}$$

- Refleja la proximidad del nodo a un obstáculo. Este índice he probado mediante experimentos que no funciona bien solo y que necesita conocer el número máximo de vecinos que puede tener un nodo.

$$\frac{findNeighbor(node).size}{\max\_neighbor}$$

- Refleja la proporción del mapa explorado y requiere conocer las dimensiones del mapa.

$$\frac{\#openList + \#closedList}{dimensions}$$

Búsqueda bidireccional

En fundamento de esta idea es que el número de nodos evaluados depende del origen de la búsqueda.

La suma de caminos óptimos bajo una heurística monótona y un coste uniforme genera un camino óptimo.

Existe 4 posibles finales para esta estrategia:

- $start \rightarrow goal$: el camino que empieza desde inicio encuentra el destino y no hay que cambiar nada.
- $goal \rightarrow start$: el camino que empieza al revés encuentra el origen y hay que intercambiar los padres con los hijos.
- $start \rightarrow middle \rightarrow goal$: uno de los dos caminos encuentra un nodo de la lista de cerrados del otro y se construye un camino juntando los dos.

En los sucesivos experimentos realizados he constatado que puede reducir tanto el tiempo empleado en crear el camino, como el número de nodos explorados si se presenta el 3º caso. En el caso 1º y 2º tenemos un camino en menos tiempo, duplicando prácticamente el número de nodos explorados.

Limpieza en abiertos

Se trata de establecer un criterio para reducir el número de nodos que se incluyen en abiertos ya sea por algún todo o índice dinámico de los anteriormente mencionados.

Esta técnica no asegura el óptimo, pero reduce considerablemente el número de nodos explorados

Aceleración local

Esta técnica tiene sentido si nuestro robot acepta planes formados por puntos separados entre sí.

Consiste en un método de escalada aplicado bajo un criterio. El algoritmo sería el siguiente:

$$
\begin{aligned}
&local\_search(node, k, closed, open, goal)\,\{\\
&\quad for\,(i \in (0 \ldots k))\,\{\\
&\quad\quad neighbor \coloneqq findNeighbor(node)\\
&\quad\quad neighbor \coloneqq neighbor - (closed \cup open)\\
&\quad\quad if\ (goal \in neighbor)\ return(neighbor)\\
&\quad\quad node \coloneqq \min(\text{neighbor})\\
&\quad\}\\
&\quad return\ (neighbor)\\
&\}
\end{aligned}
$$

Crea un camino rápido y seguro hasta el objetivo si el terrero está despejado. El problema es saber cuándo aplicarlo. Creo que con lo que mejor combina es con el algoritmo $Jump\ Point\ Search$.

Tarea 3: Experimentación

A^* estándar

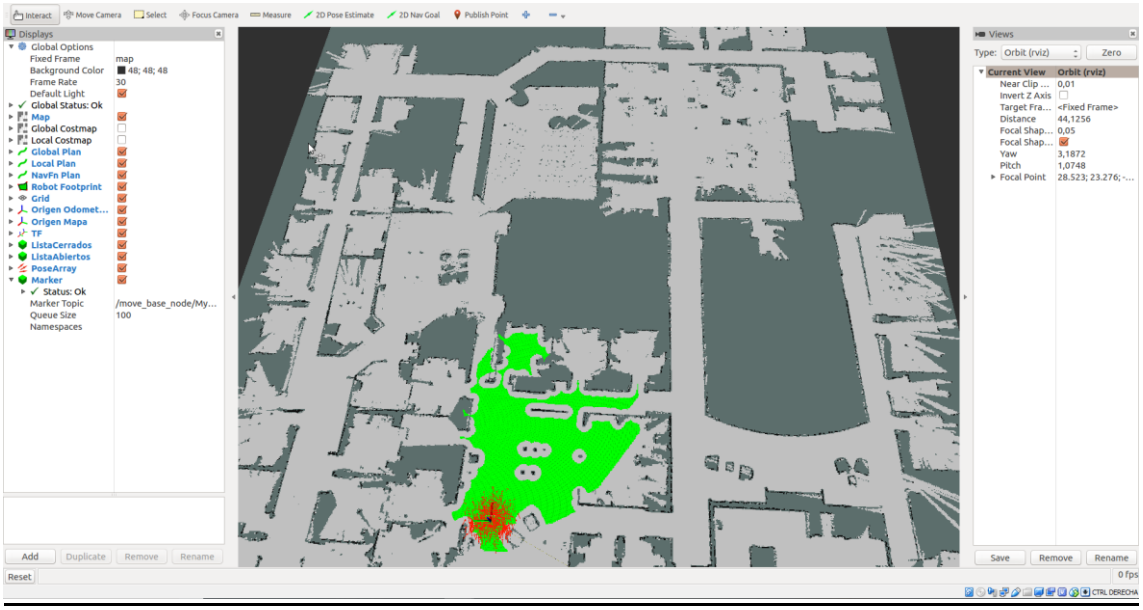

Tabla de resultados

Tiempo empleado	Indeterminado
Número de nodos explorados	Indeterminado
Tamaño del camino	Indeterminado
Distancia	Indeterminado

Haremos una comparación con el resto de técnicas cuando ni siquiera ha podido finalizar su ejecución la versión estándar.

A^* pesos

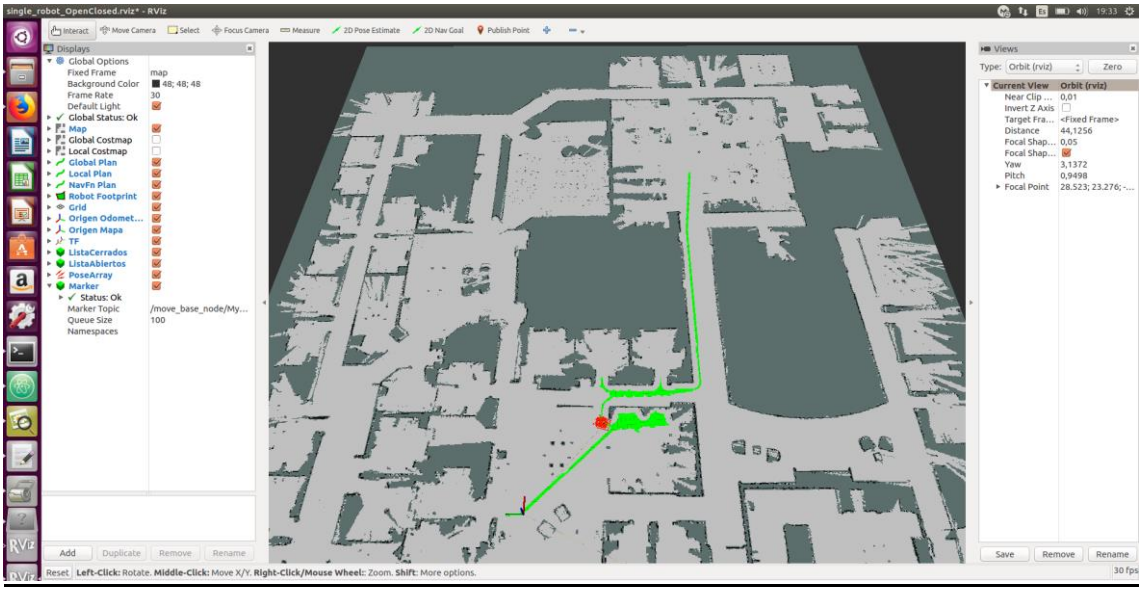

Tabla de resultados

Tiempo empleado	2.73 segundos
Número de nodos explorados	4142 nodos
Tamaño del camino	512 nodos
Distancia	27.78

*A** bidireccional

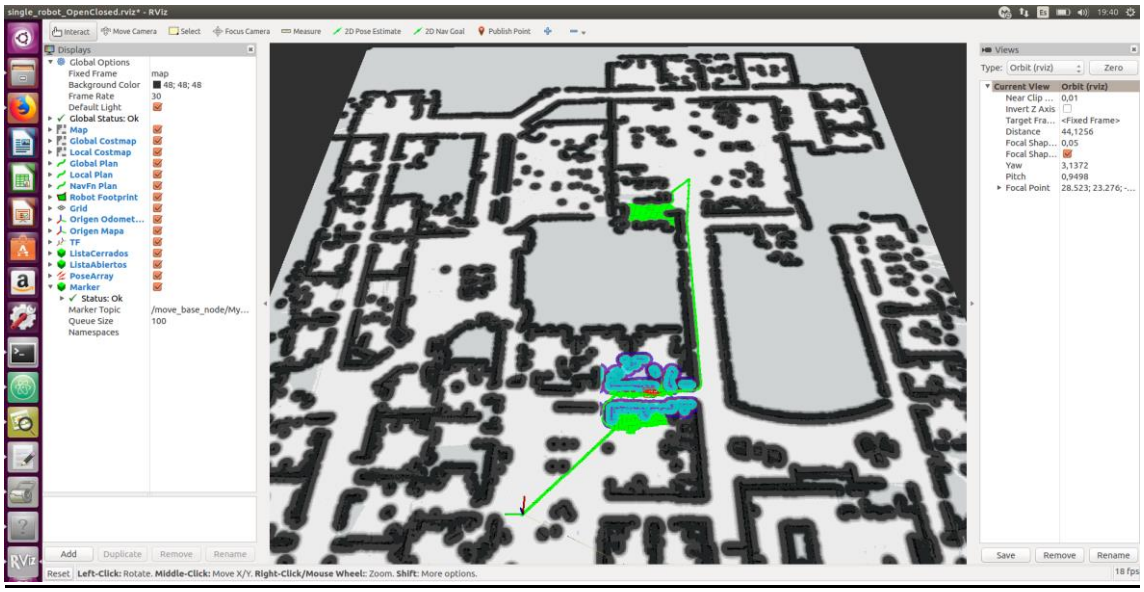

Tabla de resultados

Tiempo empleado	3.2 segundos
Número de nodos explorados	3806 nodos
Tamaño del camino	627 nodos
Distancia	34.68

*A**aceleracion local

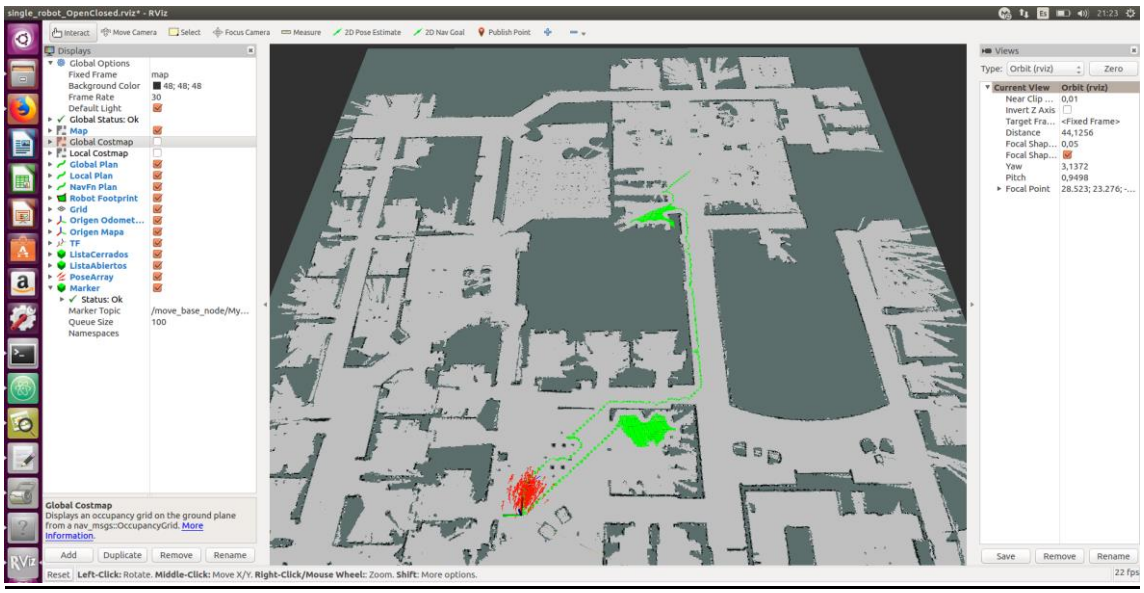

Tabla de resultados

Tiempo empleado	2.6 segundos
Número de nodos explorados	2300 nodos
Tamaño del camino	170 nodos
Distancia	37.68

Conclusión

Con el cambio de variable realizado mediante la técnica de aceleración local con valor de salto $k = 4$, es con el que consigo mejores resultados. He realizado múltiples pruebas con los índices dinámicos, pero no he llevado a una expresión generalizadora que sirva para casos variados.

Siempre que optimizamos un déficit del algoritmo A^* perjudicamos sistemáticamente otro, por no hablar de la perdida de optimalidad.