UNIVERSIDAD
POLITÉCNICA
DE MADRID

Robots autónomos

Práctica 3

Fernando Cortés Sancho

Master Universitario en Inteligencia Artificial

1 de enero de 2021

Índice

1. Introducción

El trabajo realizado en esta práctica consiste en el desarrollo del algoritmo *Rapidly-exploring Random Trees* (RTT) en una aplicación.

Para ello se ha utilizado código de una aplicación de python alojado en Github de un usuario llamado CandyZack [1], que se explicará en la sección 2.

Después, en la sección 3 se explicará el algoritmo RRT, así como su implementación en python paso a paso.

A continuación se verán los resultados de las pruebas realizadas en 3 distintos tipos de escenarios, dos de ellos propuestos por el enunciado, en la sección 4.

Por último, se extraerán conclusiones de los resultados y se propondrán líneas de mejora en la sección 5

[1] `https://github.com/candyzack/RRT-Planning-Algorithm`

2. Trabajo anterior

El trabajo del que se parte corresponde a una aplicación creada en python por el usuario CandyZack. En esta aplicación se pueden dibujar el entorno mediante círculos negros que servirán como obstáculos. Después se define el punto inicial y el punto de destino. Un ejemplo sencillo de este funcionamiento se puede ver en la Fig. 1

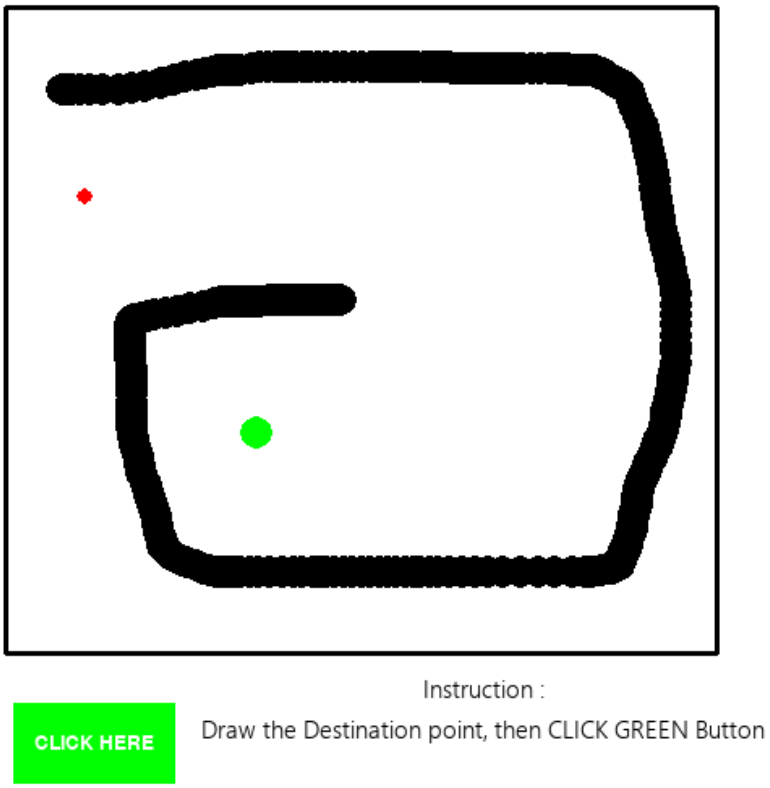

Figura 1: Aplicación del usuario CandyZack

Para ello se utiliza la famosa librería de python llamada *pygame*, que crea la ventana y controla lo que ocurre cuando se hace click dentro de ella.

La aplicación también realiza el algoritmo RRT que encuentra un camino desde el punto inicial al de destino. Sin embargo, este código no está comentado y se hace difícil de entender, por lo que este código se desechó y se decidió implementar el algoritmo desde cero, que se explicará en la siguiente sección. También se introdujo una variante que mejora la velocidad del algoritmo.

Además, se cambiaron otros aspectos. Por ejemplo, la aplicación original permitía definir varios puntos de destino, y en la aplicación final solo se puede poner un único punto.Los botones originales se mantuvieron aunque se les añadió el atributo de texto, no presente en la aplicación original, para facilitar el cambio de frase.

Por lo tanto, se podría decir que se conservan aspectos estéticos y de configuración de pygame, pero el contenido y el algoritmo son creados desde cero.

3. Algoritmo RRT

El algoritmo RRT es un algoritmo que consiste en crear aleatoriamente nodos de un árbol que crece incrementalmente hasta encontrar la meta, creando así un camino desde el punto inicial definido hasta ella.

Para ello, se parte del punto inicial y se crea un nodo en cada iteración de un bucle que no para hasta encontrar la meta. Para añadir cada nodo se crea un nodo random en el entorno y se buscar el nodo mas cercano, donde se creará el nuevo nodo a una distancia delta predefinida y en la dirección del nodo random. Este algoritmo se puede ver en forma de pseudo código en la Fig. 2, sacado de la Wikipedia [2]

```
Algorithm BuildRRT
    Input: Initial configuration q_init, number of vertices in RRT K, incremental distance Δq)
    Output: RRT graph G

    G.init(q_init)
    for k = 1 to K do
        q_rand ← RAND_CONF()
        q_near ← NEAREST_VERTEX(q_rand, G)
        q_new ← NEW_CONF(q_near, q_rand, Δq)
        G.add_vertex(q_new)
        G.add_edge(q_near, q_new)
    return G
```

Figura 2: Algoritmo RRT.

A continuación se irá explicando con código de python cada paso del algoritmo.

En la Fig. 3 se puede ver el algoritmo RRT implementado. Como se puede ver, primero se crea un nodo random que se encuentre dentro del rectángulo en entorno, de la forma en la que se puede ver en la Fig 4.

Después se mira si ese nodo se encuentra en un obstáculo o no, observando si el pixel en el que se encuentra tiene el color negro o no. Esta función en la que se puede ver en la Fig. 5.

A continuación se calculan todas las distancias al nodo random de todos los nodos para ver cual de ellos es el mas cercano, como se puede ver en la Fig. 7. La distancia se calcula mediante la distancia euclídea de ambos puntos.

Una vez encontrado el nodo mas cercano se crea el nuevo nodo en la dirección del nodo random. Si el nuevo nodo está dentro de los límites del entorno y no se encuentra en un obstáculo, se añade al grafo se guarda en un diccionario que el padre del nuevo nodo es el nodo mas cercano. Esto servirá mas tarde para establecer el camino desde la inicio hasta el final.

Después viene la comprobación de si el nuevo nodo se encuentra ya en la meta o no. Para ello se utiliza la función que se puede ver en la función de la Fig. 7. Esta comprobación, de la misma manera que con la comprobación de obstáculos, se hace fijándose en el color del píxel, en este caso verde.

[2] http://en.wikipedia.org/wiki/Rapidly_exploring_random_tree

```
while running:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
            break

    random_node  = random_point()
    if is_is_obstacle(random_node):
        continue
    nearest_node = Nearest(G,random_node)
    new_node = New_node(nearest_node,random_node)
    if not is_is_obstacle(new_node) and is_inside_game(new_node[0],new_node[1]) and new_node not in parent:
        G.add_node(new_node)
        parent[new_node] = nearest_node
        if not goal_reached(new_node):
            pygame.draw.circle(screen, blue, (new_node), 2)
            pygame.draw.line(screen, blue, (nearest_node), (new_node), 2)
            pygame.display.update()
        else:
            pygame.draw.circle(screen, blue, (new_node), 2)
            pygame.draw.line(screen, blue, (nearest_node), (new_node), 2)
            pygame.display.update()
            running = False
            lastnode = new_node
            break
```

Figura 3: Algoritmo RRT implementado en python

```
def random_point():
    random_x = random.randint(x_, x_ + 440  - 1)
    random_y = random.randint(y_, y_ + 400  - 1 )
    return (random_x, random_y)
```

Figura 4: Creación de un nodo random en el entorno

En el caso en el que el nodo no haya llegado a la meta, se dibuja y se continúa en el bucle, para seguir incrementando el árbol. Por otro lado, en el caso en el que sí que se encuentre en la meta, se guarda este nodo como último nodo, se acaba el bucle y se procede a dibujar el camino correspondiente.

Esto se realiza mirando el diccionario de padres, que ha ido guardando durante las iteraciones el padre de cada nodo, por lo que se debe ir desde el último nodo hacía atrás, remarcando los enlaces entre estos, de la manera en la que se puede ver en la Fig. 8.

```python
def is_is_obstacle(point):
    x,y = point[0],point[1]

    if screen.get_at((x,y)) == black:
        return True
    return False
```

Figura 5: Comprobación de si el punto se encuentra en un obstáculo del entorno.

```python
def dist(p1,p2):
    return math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)


def Nearest(G,point):
    shortest_dist = 100000000000000;
    shortest_node = (0,0)
    for node in G.nodes():
        distance = dist(node,point)
        if distance <= shortest_dist:
            shortest_dist = distance
            shortest_node = node

    return  shortest_node
```

Figura 6: Búsqueda de nodo mas cercano al nodo random.

```python
def goal_reached(node):
    x,y = node
    if screen.get_at((x,y)) == green:
        return True

    return False
```

Figura 7: Comprobación de si el nuevo nodo se encuentra en la meta.

```
while (lastnode != Start):
    pygame.draw.circle(screen, green, (lastnode), 2)
    pygame.draw.line(screen, green, parent[lastnode], lastnode, 2)
    lastnode = parent[lastnode]
    pygame.display.update()
```

Figura 8: Generación del camino encontrado.

4. Pruebas y resultados

El algoritmo se probó en versiones en tres tipos de escenarios distintos. Los dos primeros son recomendados por el enunciado, y el último consiste en una espiral. Y se pueden ver en las Fig. 9,10 y 11.

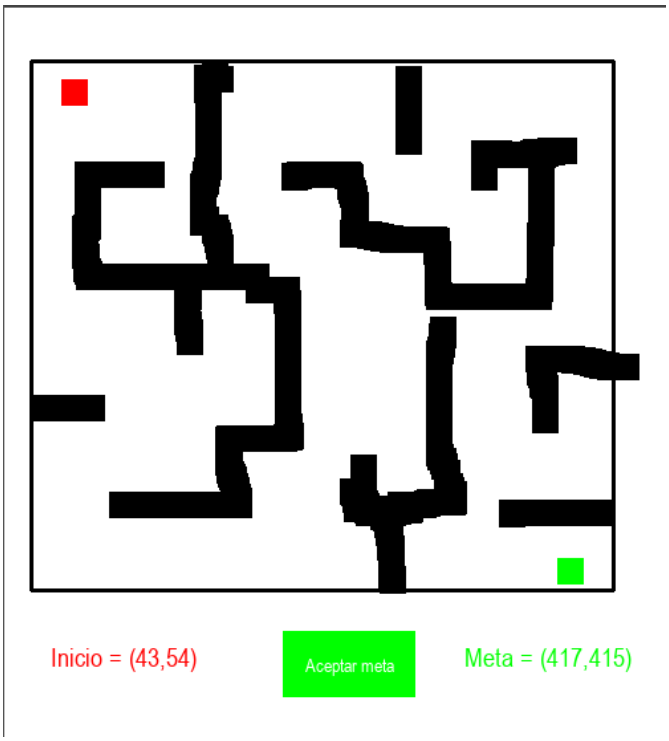

Figura 9: Entorno de pruebas número 1

La primera versión es el algoritmo original, que crea el nuevo nodo en la dirección del nodo creado aleatoriamente, con un valor máximo de distancia entre el nodo mas cercano de 10 píxeles. Esto se puede ver implementado en la Fig. 12.

La segunda versión surgió de manera fortuita al intentar implementar el algoritmo anterior, pero , como se verá luego, presenta mejores resultados. El algoritmo consiste en crear nodos a una distancia fija pero siempre en diagonal. Esto hace que se ocupen los espacios más rápidamente. En la Fig. 13 se puede ver implementado

En las Fig. 14 y **??** se puede ver el resultado de ejecutar ambas versiones en el primer entorno.

En las Fig. 16 y 17 se puede ver el resultado de ejecutar ambas versiones en el segundo entorno.

Por último, las pruebas realizadas en el tercer entorno se pueden ver en las Fig. 18 y 19

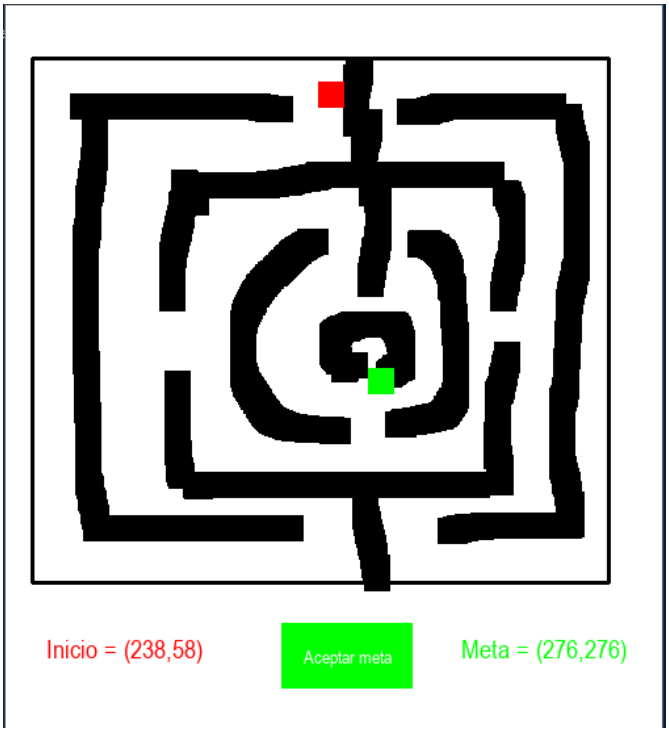

Figura 10: Entorno de pruebas número 2

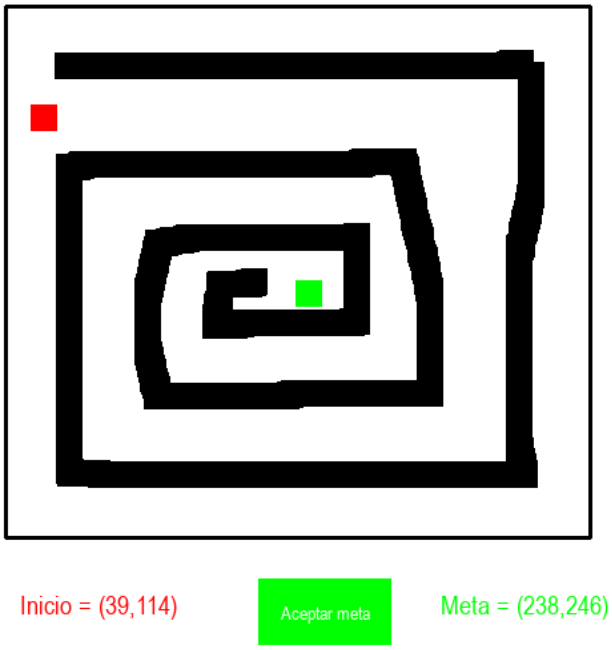

Figura 11: Entorno de pruebas número 3

```python
def New_node(nearest_node,random_node):

    s = []
    s.append(random_node[0]-nearest_node[0])
    s.append(random_node[1]-nearest_node[1])

    signos = numpy.sign(s)

    if signos[1] == -1:
        y = random.randint(-10,0)

    else:
        y = random.randint(0,10)

    x = math.sqrt(100-(y**2))

    if signos[0] == -1:
        x = x*-1

    return (int(nearest_node[0]+x) , int(nearest_node[1]+y))
```

Figura 12: Algoritmo de creación del nuevo nodo

```python
def New_node(nearest_node,random_node):
    y = random_node[1]-nearest_node[1]
    if y > 0:
        y = 1
    else:
        y = -1
    x = random_node[0]-nearest_node[0]
    if x > 0:
        x = 1
    else:
        x = -1

    new_node = (nearest_node[0]+ x*dist_x , nearest_node[1]+y*dist_y)
    return new_node
```

Figura 13: Segundo algoritmo de creación del nuevo nodo

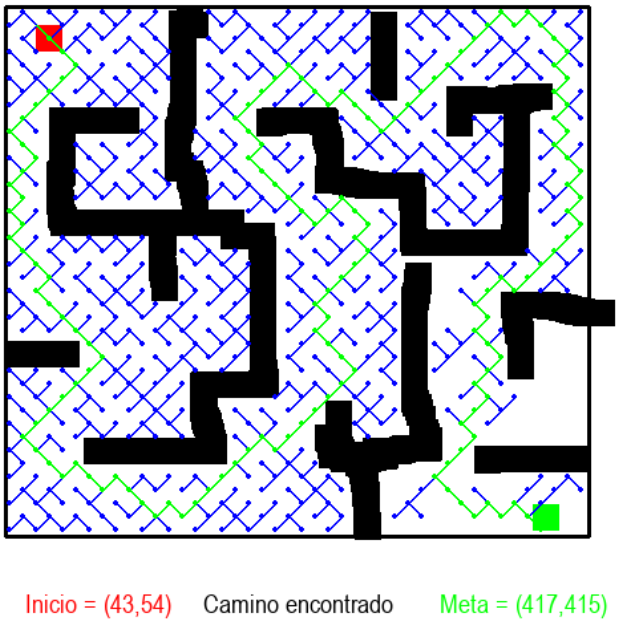

Figura 14: Resultado en el entorno de pruebas número 1

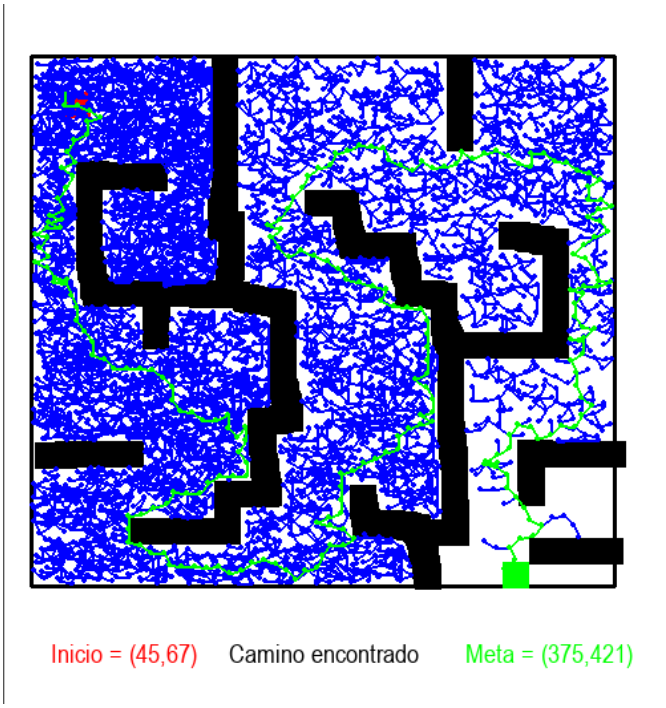

Figura 15: Resultado en el entorno de pruebas número 1

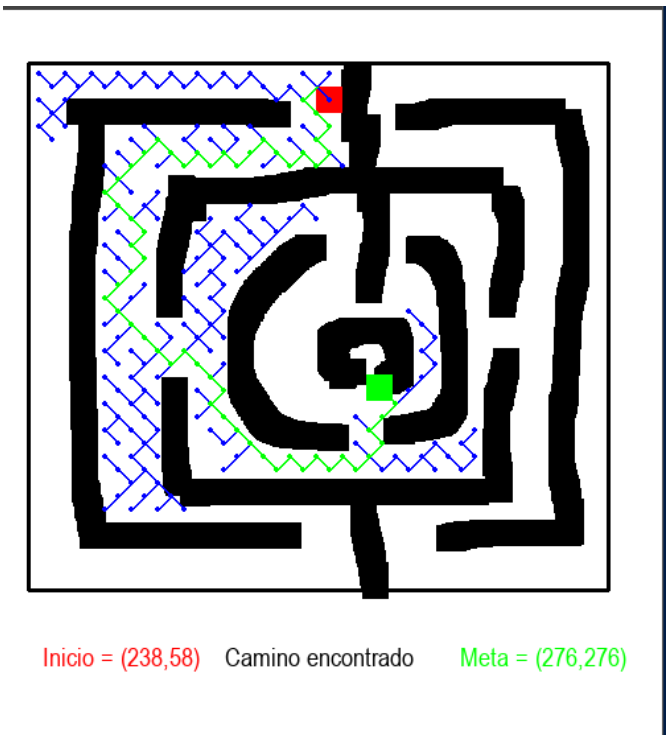

Figura 16: Resultado en el entorno de pruebas número 2

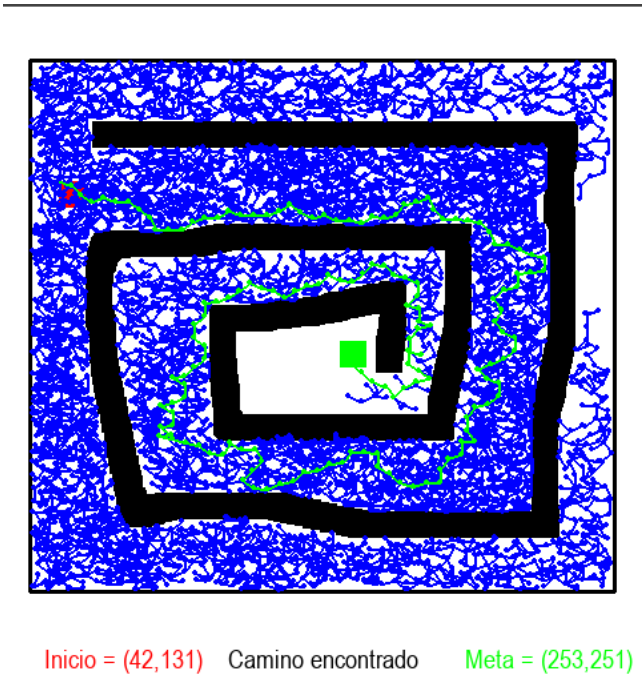

Figura 17: Resultado en el entorno de pruebas número 2

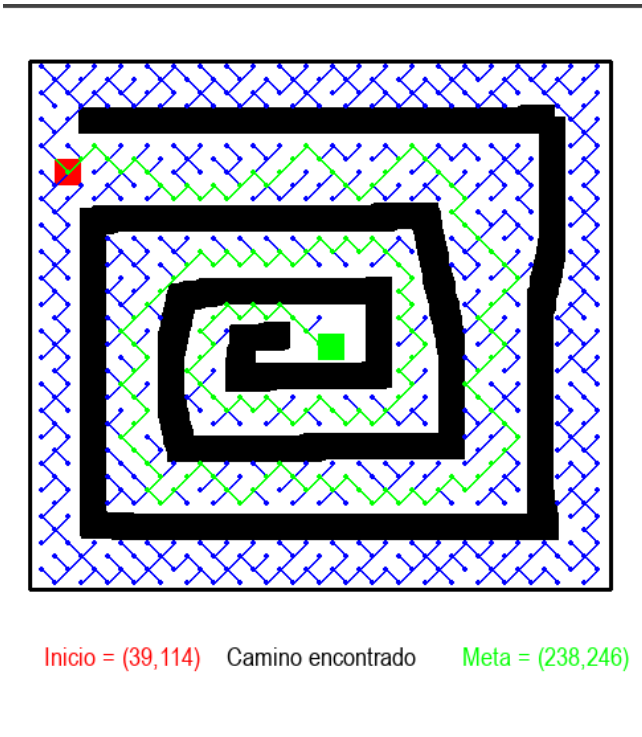

Figura 18: Resultado en el entorno de pruebas número 3

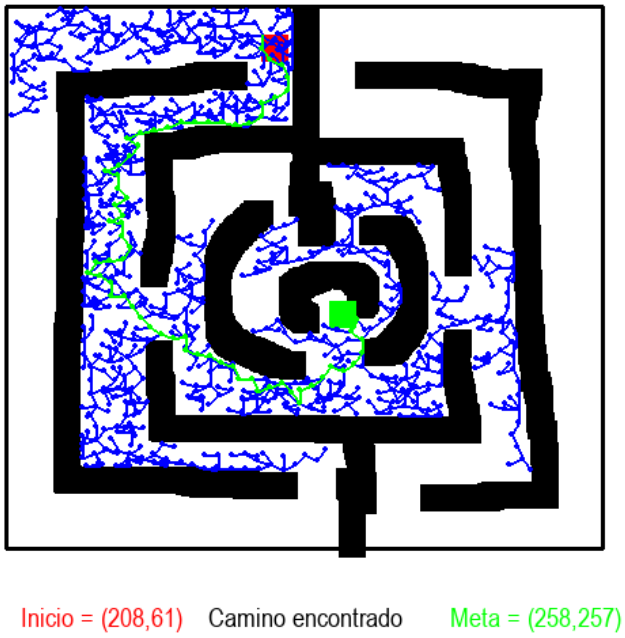

Figura 19: Resultado en el entorno de pruebas número 3

5. Conclusiones y líneas de mejora

A continuación se expondrán conclusiones de lo visto en la sección anterior y se hablará del algoritmo RRT*.

En primer lugar y como se ha podido ver en la sección anterior, al algoritmo original le hace falta mucho mas tiempo para encontrar el camino. El segundo algoritmo lo encuentra muy rápido. El espacio de búsqueda de este último esta acotado, ya que solo pueden existir nodos en determinadas posiciones, lo que hace que la exploración por el entorno sea mucho mas rápida.

Sin embargo, como se puede intuir, este algoritmo está sesgado y funciona bien ya que las dimensiones de la meta son suficiente como para que no encaje en un espacio en blanco entre nodos y no pueda ser visitado.

Por lo tanto podríamos decir que , aunque se observa un mejor resultado, se debe al sesgo que existe y que se esta tratando con un entorno en 2-d, que simplifica las cosas

Por último y a modo de propuesta de línea de mejora, existe una versión mejorada del algoritmo RRT llamado RRT*. Este algoritmo suaviza el camino, haciéndolo mas recto y por lo tanto mas corto. Cada vez que se crea un nuevo nodo, se evalúa si alguno de los nodos vecinos del nuevo nodo está mas cerca que su nodo padre. Si es así se crea un enlace entre ellos. Esto evita el *zig-zag* que existe en los caminos encontrados por el algoritmo RRT.