MÁSTER UNIVERSITARIO EN INTELIGENCIA ARTIFICIAL

Robots Autónomos

Práctica 3: Representaciones del espacio y planificación

Javier Hermida Lario

1. ÍNDICES

Índice de figuras

2. INTRODUCCIÓN

A la hora de automatizar el movimiento de un robot autónomo, es crucial la manera en la que se representa el espacio y a su vez, cómo es utilizado para planificar una ruta válida y eficiente. Este dominio de problemas es sumamente amplio debido a la gran variabilidad de situaciones tanto en el espacio (2D, 3D, terrestre, aéreo, etc.) como en las cualidades del robot (holonómico, no holonómico, con autonomía limitada, etc.). Además, existen un gran número de algoritmos distintos para realizar su planificación una vez se ha representado el espacio.

1.1. Objetivos

Dentro de este contexto, este trabajo tiene como objetivo principal implementar un algoritmo de planificación dentro de un escenario con la restricciones de tratarse de un mapa de dos dimensiones, con una representación continua del espacio donde se simula un robot sin volumen y holonómico. Para alcanzar esto se han planteado una serie de objetivos intermedios:

- Diseñar el algoritmo e implementarlo.
- Crear una representación visual del escenario y la solución aportada por el algoritmo.
- Probarlo con distintos casos de prueba

1.2. Material entregado

Junto con esta memoria, el material entregado consta de una serie de imágenes que representan los distintos casos de prueba, y por otro lado, dos ficheros de Python que contienen el algoritmo de búsqueda implementado y el planificador al completo.

1.3. Estructura del documento

La estructura de este documento es la siguiente: en primer lugar se desarrolla el diseño del algoritmo de planificación y sus distintas partes. Seguidamente se discuten y muestran los distintos casos de prueba comparando las distintas aproximaciones. Finalmente, se desarrollan las conclusiones obtenidas del desarrollo del algoritmo y las pruebas realizadas con él. Adicionalmente se comenta en un anexo las instrucciones de ejecución del código aportado junto a esta memoria.

3. DISEÑO DEL ALGORITMO

En este apartado se desarrolla el diseño del algoritmo de planificación implementado y las partes que lo conforman de manera individual, así como las distintas maneras de implementarlas.

3.1. Algoritmo implementado

El algoritmo escogido se enmarca en la familia de los *Probabilistic Road Maps* (PRM). Estos algoritmos se caracterizan por ser utilizados en entornos cuyo espacio de configuración es demasiado complejo para algoritmos de planificación tradicionales. Se basan en crear caminos en los que se asegura que el robot no colisiona con los obstáculos de una manera probabilística. Existen diversas maneras de implementar estos algoritmos pero normalmente se encuentran dos partes diferenciadas: la fase de aprendizaje, en donde se genera un grafo de forma aleatoria que representa una serie de caminos donde el robot no colisiona con ningún obstáculo, y la fase de *query* donde se une ese grafo con el punto de inicio y el de final y se utiliza un algoritmo de búsqueda para generar una ruta en caso de que existan. A continuación se detalla cómo se implementan estas fases específicamente.

3.2. Fase de aprendizaje

En la fase de aprendizaje se realizan dos pasos principales: la generación de puntos aleatorios que conforman los nodos del grafo y el proceso de unirlos mediante aristas que no colisionan.

3.2.1. Generación de puntos del grafo

Para la generación de los puntos se han estudiado distintas aproximaciones, las cuales tienen distintos beneficios e inconvenientes que se discuten a continuación para cada una de ellas.

3.2.1.1. Generación en *grid* cuadrada

Esta generación se realiza generando una serie de puntos en forma rectangular que están separados vertical y horizontalmente lo mismo entre ellos. Como parámetro se le introduce el tamaño de esas separaciones. Las ventajas que tiene son su simpleza, la capacidad de tener una buena cobertura ajustando el tamaño de las separaciones y su buen desempeño en escenarios donde los obstáculos son paralelos a los ejes. Sin embargo, también presenta el inconveniente de ser muy poco eficiente cuando los obstáculos son circulares o presentan formas no cuadradas. Un ejemplo se puede observar en la Fig. 3.1.

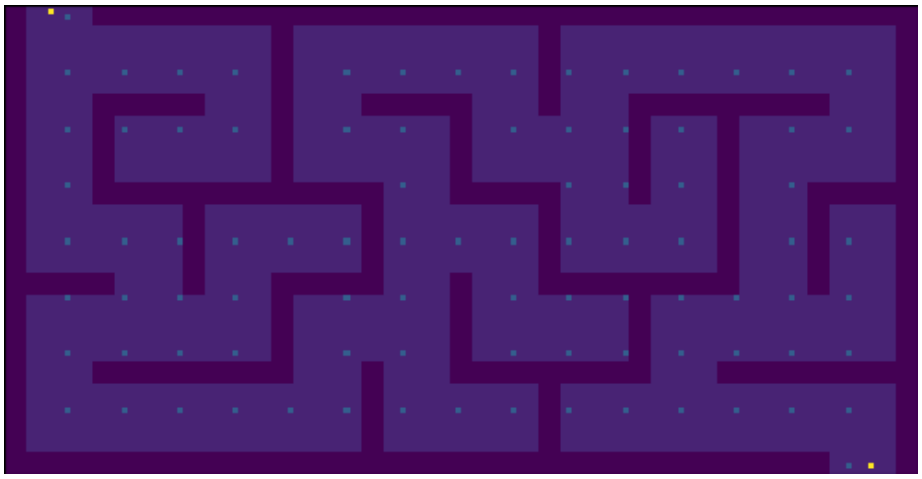

Fig. 3.1. *Ejemplo de generación en grid cuadrada*

3.2.1.2. Generación completamente aleatoria

En este tipo de generación se obtienen los puntos de manera completamente aleatoria de entre todos los puntos posibles del entorno. Como parámetro se especifica el número de puntos a ser generados. Las ventajas de esta manera de obtener los puntos del grafo es su simpleza que se ve reflejada en el poco tiempo que lleva generarlos. Como desventaja se encuentra la posibilidad de obtener poca cobertura del mapa debido a que los números aleatorios no se hayan generado uniformemente, lo cual suele pasar cuando la cantidad es demasiado baja o el mapa demasiado grande. Un ejemplo de este tipo de generación se puede observar en la Fig. 3.2.

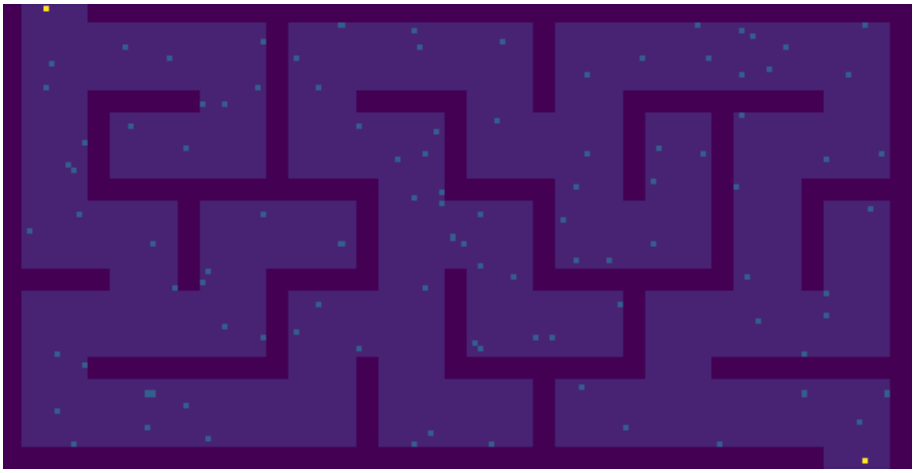

Fig. 3.2. *Ejemplo de generación completamente aleatoria*

3.2.1.3. Generación aleatoria en sectores cuadrados

Este tipo de generación es una mezcla de las dos anteriores y genera sus puntos en primer lugar dividiendo el entorno en sectores cuadrados, y después generando dentro de cada uno de esos cuadrantes una serie de puntos aleatorios. Este algoritmo combate la flaqueza de la generación en *grid* de ser débil frente a obstáculos no cuadrados, ya que se

trata de una generación aleatoria, y también hace frente al inconveniente de la poca cobertura de la generación completamente aleatoria ya que utiliza los cuadrantes asegurando que hay puntos en todos los sectores. Su mayor inconveniente es que el coste computacional es mayor que los otros dos métodos. Como parámetros recibe el tamaño de los cuadrantes y el número de puntos aleatorios a generar en cada uno de ellos. Un ejemplo se puede observar en la Fig. 3.3.

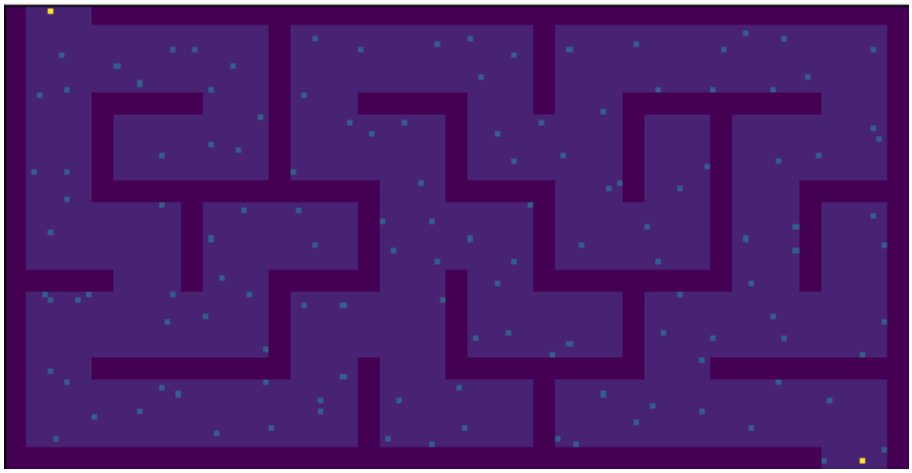

Fig. 3.3. *Ejemplo de generación aleatoria por sectores cuadrados*

3.2.2. Unión de los puntos del grafo

Una vez se han generado los puntos que conforman el grafo encargado de representar las posibles rutas a tomar por el robot, es necesario saber cuáles de ellos pueden ser unidos y por lo tanto pueden contener un segmento de la ruta. Dado que los algoritmos PRM se centran en detectar si el robot colisiona con los obstáculos y que además se considera que el diámetro del robot es cero, se puede comprobar si dos puntos se pueden unir si al trazar una recta entre ellos esta no colisiona con ningún obstáculo.

Un inconveniente a la hora de realizar estas comprobaciones es el hecho de que estas colisiones se dan en un espacio de coordenadas continuo mientras que la representación del escenario y su representación visual son discretos. Es por ello por lo que se tiene que hacer una discretización de las rectas que se generan, una aproximación que en caso de ser muy tosca puede generar problemas, como no detectar ciertos muros finos o crear rectas discontinuas en la representación visual.

Una vez solucionado el problema de la discretización, otro desafío es la selección de aquellos puntos que se tienen que unir entre sí. La respuesta más fácil a esta pregunta es unir cada punto con todos los demás. Esta aproximación tiene una ventaja destacada y es que a la hora de calcular la ruta, siempre se obtendrá el camino más corto posible para

eso nodos en concreto. Sin embargo, comprobar la unión de estos puntos de esa manera es un problema de complejidad cuadrática y que también aumenta con el tamaño del entorno, lo que supone que cuando el número de puntos aumenta, el tiempo de ejecución empieza a ser muy prolongado.

Otra alternativa es utilizar un vecindario a la hora de unir los puntos y solamente realizar las comprobaciones de colisión cuando estos puntos cumplen unas condiciones de cercanía. De esta manera se consigue reducir drásticamente la complejidad del algoritmo. Sin embargo, este método producirá normalmente rutas más costosas y puede darse el caso, en el que dependiendo de la generación de puntos, se cree un grafo inconexo al no cumplirse las condiciones del vecindario. En la implementación se utiliza un vecindario de distancia euclídea, donde dos puntos se unen solamente si se encuentran a menos de una distancia especificada. Este tipo de vecindario puede funcionar bien juntamente con la generación de puntos aleatoria por cuadrantes, ya que asegura la cobertura de los puntos.

3.3. Fase de *query*

Una vez construido el grafo de la fase de entrenamiento, se realiza la fase de *query* en la cual, en primer lugar, se intentan conectar el nodo de inicio y el de final con el grafo, y posteriormente se aplica un algoritmo de búsqueda de grafos para intentar obtener un camino que una el inicio y el final, a la vez que se intenta minimizar el coste.

En este trabajo se ha implementado como algoritmo de búsqueda un A* el cual utiliza como heurística la distancia euclídea entre dos puntos. Esta heurística es admisible por lo que siempre que exista, la solución obtenida será la óptima para el grafo que se genera en el entrenamiento. Con respecto a su coste computacional, este es despreciable con respecto a la fase de aprendizaje por lo que no es necesario estudiarlo en profundidad.

4. CASOS DE PRUEBA

Para poder evaluar los resultados aportados por el planificador se plantean una serie de casos de prueba, donde se utilizan los distintos métodos para generar puntos y calcular los vecinos de cada nodo. Estos se muestran a continuación.

4.1. *Maze* 1

El primer *maze* es el más sencillo de todos teniendo unas dimensiones de 186x64 pixeles y se puede observar su estructura en la Fig. 4.1. Este destaca por no tener pasillos muy alargados y tener una estructura rectangular.

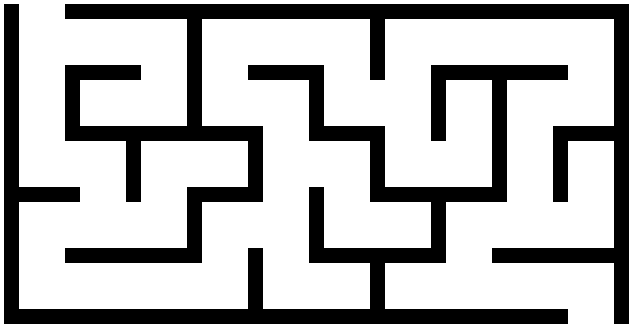

Fig. 4.1. *Maze 1*

Al tratarse de generaciones aleatorias no tiene sentido comparar los resultados entre métodos de generación de puntos ya que pueden producir rutas totalmente diferentes solamente por ese factor aleatorio. Sin embargo sí es interesante comparar como el método de elección del vecino altera los resultados. Por ejemplo, en las Fig. 4.2 y Fig. 4.3 se muestran dos ejecuciones con y sin vecindario respectivamente.

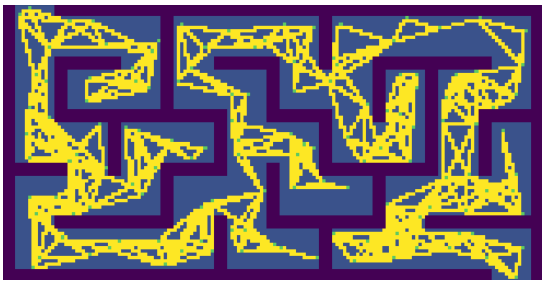

Fig. 4.2. *Grafo Maze 1 con vecindario*

Fig. 4.3. *Grafo Maze 1 sin vecindario*

En la que utiliza vecindario, la ejecución lleva 1.81 segundos y produce una ruta de 418.5 de coste. Por otro lado aquella que no utiliza vecindario con el mismo método de generación de puntos lleva 24.01 segundos y produce una ruta de 407 de coste. La diferencia entre costes es muy reducida y puede estar causada por la aleatoriedad de la generación de puntos, sin embargo, el coste de tiempo es mucho mayor dado que su

complejidad también lo es. Un ejemplo de ruta generada para este problema en concreto se muestra en la Fig. 4.4.

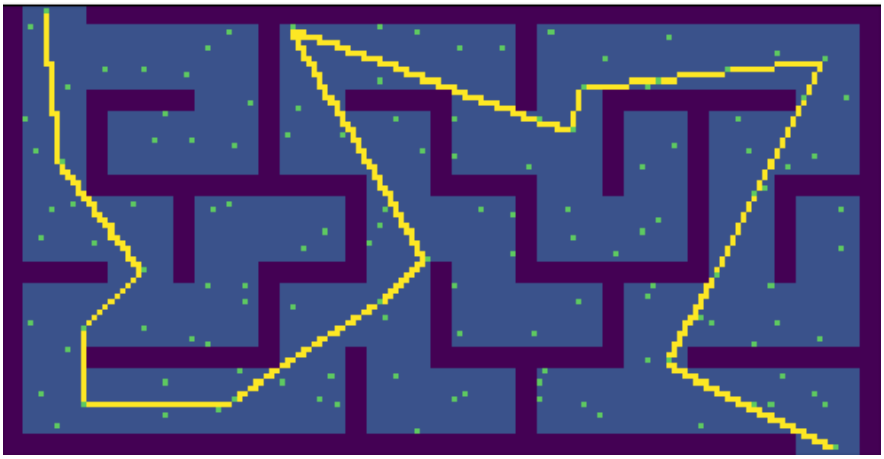

Fig. 4.4. *Ejemplo de path Maze 1*

4.2. *Maze* 2

El segundo de los casos de prueba se trata de un *maze* de 430x430 píxeles que se caracteriza por las formas curvas que presenta en el centro y por los fragmentos sin salida. Este se muestra en la Fig. 4.5.

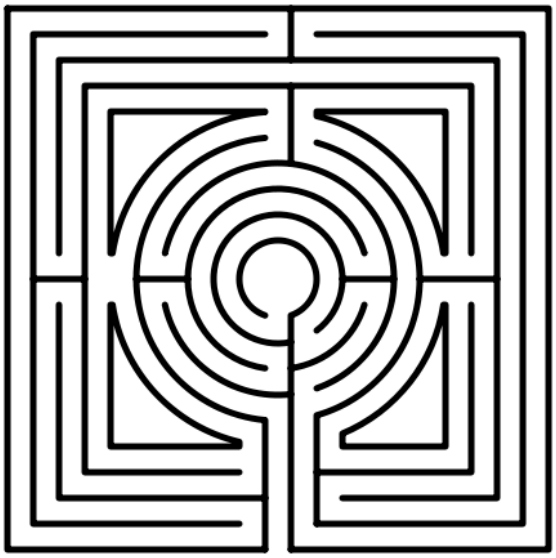

Fig. 4.5. *Maze 2*

Este caso pone más a prueba al planificador ya que las dimensiones son mucho mayores pero manteniendo caminos estrechos y curvas cerradas. En las Fig. 4.6 y Fig. 4.7 se muestran los caso con y sin vecindario respectivamente y se observa como el caso sin vecindario tiene un grafo mucho más denso ya que une todos los puntos que puede con todos.

Fig. 4.6. *Grafo Maze 2 con vecindario*

Fig. 4.7. *Grafo Maze 2 sin vecindario*

Utilizando vecindario se obtiene en 29.31 segundos una solución de coste 4664.58. El caso sin vecindario tarda 22 minutos y genera una solución de coste 4678.67. La diferencia de coste es despreciable pero el coste temporal es drásticamente diferente, siendo para el caso sin vecindario mucho más elevado y siendo inviable para un planificador que requiera poco tiempo de ejecución. También se aprecia como en ambas se realiza una generación inútil dentro de figuras cerradas, las cuales suponen un callejón sin salida pero que se intentan unir y gastan recursos igualmente. Una de las rutas generadas como solución es la que se muestra en la Fig. 4.8**,**

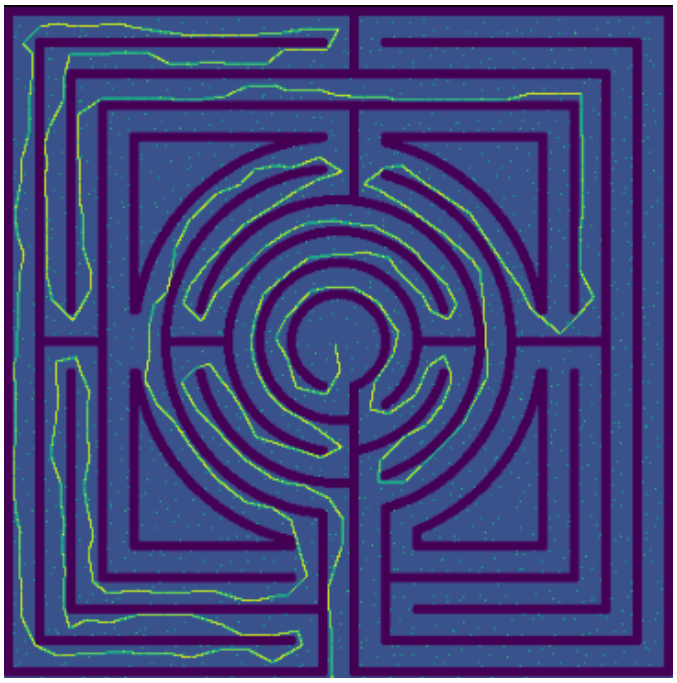

Fig. 4.8. *Ejemplo de path Maze 2*

4.3. *Maze* 3

El último de los casos de prueba se trata de un *maze* de 430x440 pixeles que se caracteriza por sus pasillos largos y anchos, lo cual se espera que genere problemas en el caso sin vecindario ya que sufre computacionalmente en este tipo de estructuras ya que generan gran cantidad de rectas. La distribución de este caso de prueba se puede observar en la Fig. 4.9**.**

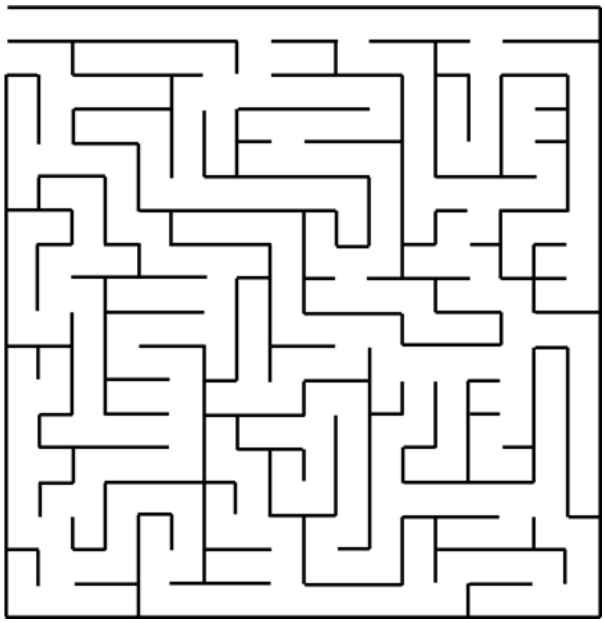

Fig. 4.9. *Maze 3*

Al igual que en el resto de casos de prueba, se utilizan una aproximación con y otra sin vecindario, los cuales se muestran respectivamente en las Fig. 4.10 y Fig. 4.11.

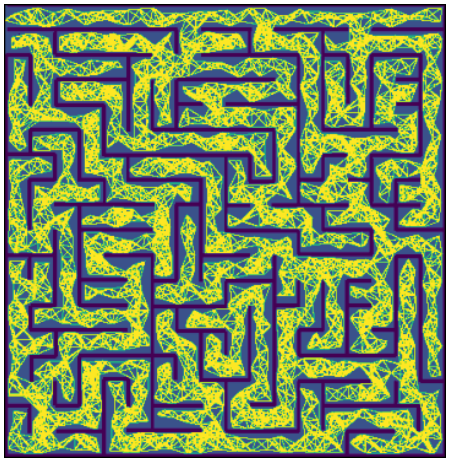

Fig. 4.10. *Grafo Maze 3 con vecindario*

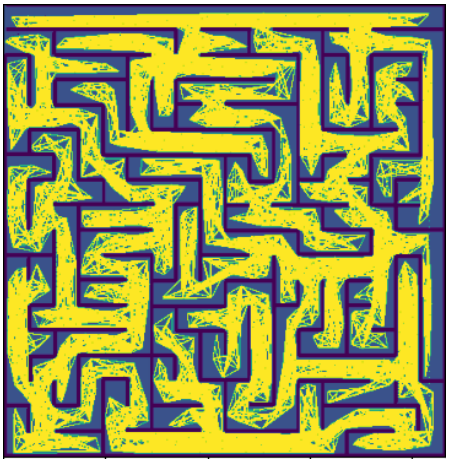

Fig. 4.11. *Grafo Maze 3 sin vecindario*

En el caso con vecindario se obtiene una solución en 39.83 segundos y con un coste de 1901.65, mientras que para el caso sin vecindario la solución se obtiene en 26 minutos

y con un coste de 1897.23. Al igual que en el caso anterior el tiempo que tarda en obtener una solución en el caso sin vecindario es mucho mayor que el caso en el que lo usa, lo cual pone de manifiesto la gran diferencia en complejidad de ambas aproximaciones. Un ejemplo de ruta obtenida para este caso de prueba es la mostrada en la Fig. 4.12.

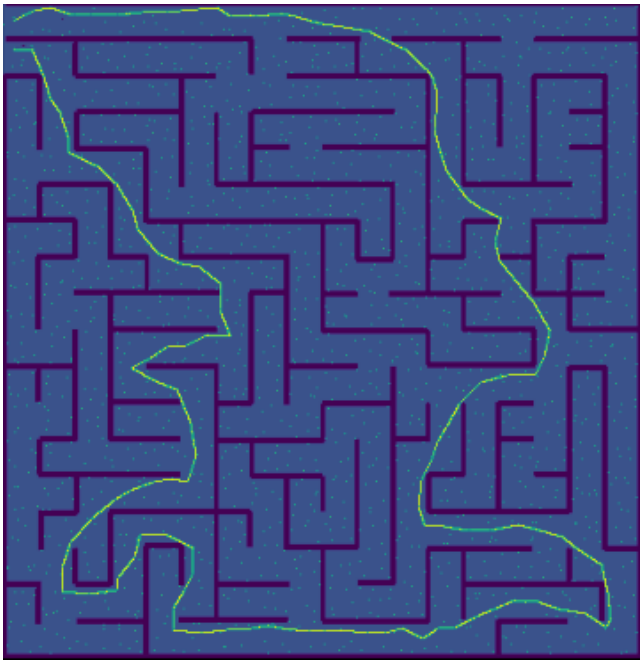

Fig. 4.12. *Ejemplo de path Maze 3*

4. CONCLUSIONES

Tras ejecutar las pruebas especificadas se obtienen una serie de conclusiones tanto acerca de los algoritmos PRM como de las distintas aproximaciones que se han probado. En primer lugar con respecto a los PRM, se observa como la aleatoriedad ofrece la ventaja de reducir drásticamente la complejidad de la planificación, pero sin embargo hace al algoritmo poco consistente. Esta baja consistencia se produce porque al generar aleatoriamente los puntos del grafo, si se generan pocos puntos o los que son generados no están distribuidos uniformemente se crea un grafo inconexo y entonces no se encuentra la solución.

Otra desventaja del algoritmo PRM implementado en específico para este trabajo, es la alta dependencia de ajustar los parámetros para los distintos casos de prueba, ya que a pesar de haber métodos que se ajustan al tamaño, hay ciertas configuraciones que necesitan de ajustes específicos. Es por ello por lo que existen otros algoritmos que suplen esta carencia, como pueden ser los *Exploring Random Trees*.

Con respecto a la generación de puntos, la más flexible de todas es la generación aleatoria por sectores ya que ofrece la mayor cobertura. Además, funciona bien con los vecindarios ya que asegura que en un cierto recinto va a haber mínimo una cantidad de puntos por lo que resuelve el problema de los vecindarios de empeorar la conectividad del grafo. Todas estas ventajas las obtiene a cambio de muy poco coste computacional por lo que es la mejor aproximación de las tres en cuanto a la generación de puntos.

Por último el aspecto más importante en términos de viabilidad del algoritmo de planificación es la manera de crear las conexiones del grafo. La manera más consistente en cuanto a obtener un grafo conexo y con la mejor ruta posible es no utilizar ningún vecindario, sin embargo, esto supone una complejidad cuadrática que además se incrementa linealmente también con el tamaño del mapa, lo que provoca que la planificación requiera de una gran cantidad de tiempo y recursos. Este aspecto hace que esta alternativa no sea viable en un caso real donde es necesario que los cálculos no tarden horas o días en realizarse. La otra aproximación es utilizar vecindarios, los cuales presentan una clara desventaja: en caso de que la generación de puntos no cree puntos lo suficientemente cerca como para estar en el vecindario, el grafo quedará inconexo a pesar de poder serlo. Es por esto por lo que esta metodología funciona bien con la generación aleatoria en cuadrantes, y también por lo que se pueden utilizar técnicas mixtas de

vecindario y no vecindario para aquellos nodos que no se conecten. Por otro lado las ventajas obtenidas en términos computacionales son inmensas y hace que estos algoritmos sean viables en entornos reales.

En resumen, de todas las metodologías experimentadas, la que utiliza vecindario y una generación aleatoria en cuadrantes de puntos, es la que más potencial presenta y produce un mejor balance entre eficiencia y coste computacional. Como posibles trabajos futuros se propone modificar el algoritmo para obtener los parámetros automáticamente del entorno y que no sean dependiente de su ajuste manual.

5. ANEXO A: INSTRUCCIONES DE USO

El código entregado está desarrollado en Python 3.10 y requiere de una serie de dependencias y variables para ejecutarlo correctamente. En primer lugar se utilizan una serie de librerías externas para utilizar funcionalidades ya creadas que faciliten el trabajo. Estas son: time, numpy, matplotlib, random y sys, las cuales es necesario instalar en caso de no tenerlas.

El directorio entregado consta de una estructura específica que es necesario mantener ya que cualquier cambio produciría un fallo en su ejecución. Esta estructura se puede observar en la Fig. A.1**.** En ella se observa el fichero Python "Main.py" el cual contiene las funcionalidades del planificador y que el fichero que se ejecuta por el usuario, el fichero Python "A_Star.py" que contiene el algoritmo de búsqueda A* que utiliza el planificador, y por último la carpeta "Mazes" que contiene las imágenes de los casos de prueba.

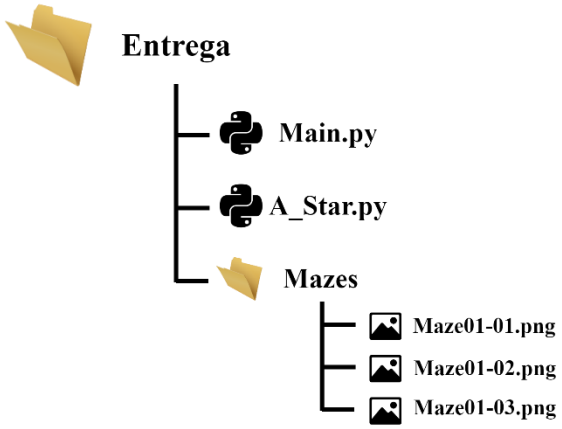

Fig. A.1. *Estructura del directorio*

Ya de cara a ejecutar el planificador es necesario ejecutar el fichero de Python "Main.py", el cual ira generando *plots* de los laberintos, los cuales hay que ir cerrando para ir avanzando en los pasos del algoritmo (generación de puntos, creación del grafo y búsqueda de la ruta óptima). La ejecución se puede hacer para tres casos de prueba distintos, con tres generaciones de puntos distintas y con 2 maneras de unir los puntos en el grafo. Para especificar qué *maze* se quiere utilizar, se utilizan las líneas de la 146 a la 159 donde es necesario descomentar el caso que se quiere utilizar y comentar el resto, como se observa en el ejemplo de la Fig. A.2.

```
# Variables para maze 1
'''maze_path = 'Maze01-01.png'
inicio = (8, 82)
final = (155, 1)'''

# Variables para maze 2
'''maze_path = 'Maze01-02.png'
inicio = (212, 0)
final = (213, 213)'''

# Variables para maze 3
maze_path = 'Maze01-03.png'
inicio = (8, 430)
final = (8, 410)
```

Fig. A.2. *Ejemplo de selección de maze*

Para seleccionar el tipo de generación aleatoria de puntos se sigue el mismo concepto que para los casos de prueba, es necesario descomentar la que se quiere utilizar y comentar el resto. Para cada una de estas generaciones existen una serie de parámetros que se explican en la sección *3.2.1. Generación de puntos del grafo.* Un ejemplo de estas líneas que son de la 168 a la 170 se muestra en la Fig. A.3.

```
#points = generate_points_grid(width,height,10)
#points = generate_points_random(width, height,100)
points = generate_points_grid_random(width, height, 1, 10)
```

Fig. A.3. *Ejemplo de selección de las distintas generaciones de puntos*

Por último se encuentra la elección de utilizar vecindario en las líneas de la 175 a la 177, donde hay una variable booleana que especifica si se usa vecindario (*True*) o no (*False*) y otra que especifica su tamaño que por defecto es 20. Un ejemplo se muestra en el código de la Fig. A.4**.**

```
# Diferentes tipos de vecindario y su tamaño
vecindario = True
size_vecindario = 20
```

Fig. A.4. *Ejemplo de selección de vecindario*