USC
UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
ESCOLA POLITÉCNICA SUPERIOR DE ENXEÑARÍA

Memoria práctica 2
Agentes Inteligentes

Losada Álvarez Adrián
Seoane Temprano Pablo

TABLA DE CONTENIDO

1. INTRODUCCIÓN

En este documento se explicará de forma detallada la resolución del problema planteado empleando el **algoritmo estrella (A*)**.

El problema que se nos plantea trata de 2 robots limpiadores que se mueven por una habitación (*ver Fig. 1*) en la que hay distintos elementos (obstáculos, puntos de carga y vaciado, etc.) y éstos deben de desarrollar automáticamente un recorrido eficiente para limpiarlas empleando A*.

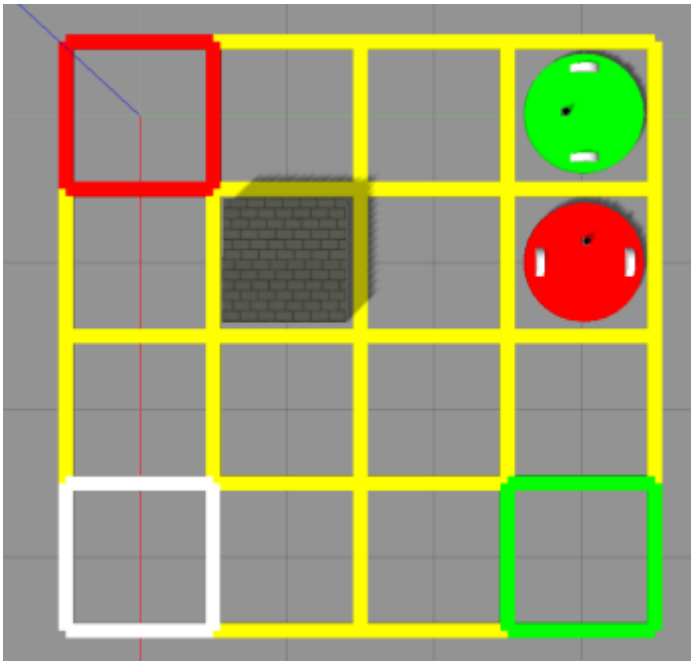


Figura 1. Forma de la habitación a limpiar

Hay que destacar que los movimientos de los robots son bastante limitados, ya que las únicas acciones que pueden realizar son:

- Avanzar: El robot avanza 1 casilla
- Girar Izq: El robot gira en sentido antihorario 90º
- Girar Der: El robot gira en sentido horario 90º

Ambos robots realizarán **solamente una** de las acciones descritas anteriormente por turno.

Por otro lado, los robots cuentan con una batería que se va gastando, dependiendo de los movimientos que realice (mov. avance = -1, mov. giro = -0.5) y deberá de volver al punto de carga (casilla roja para el robot rojo, casilla verde para el robot verde) para recargarla al máximo (carga máx = 10), además de un depósito de suciedad que se llenará a medida que los robots visiten casillas sucias (casillas amarillas) y deberán de vaciar en la casilla de vaciado (casilla blanca) cuando se llene.

Para poder implementar dicho algoritmo es necesario definir los estados iniciales y sus operadores, con sus correspondientes precondiciones y efectos, que usará para generar todos los nodos sucesores posibles para cada estado y posteriormente elegir una sucesión de nodos que nos lleve a una sucesión óptima del problema.

A continuación, se explicará en detalle cada uno de estos estados, operadores, precondiciones y efectos en detalle.

2. ESTADOS

Para definir los estados se optaron por las siguientes variables:

- *self.battery* -> Contiene el valor de batería actual de cada robot (se inicializa al máximo).

- *self.vacum* -> Contiene el valor de depósito vacío restante de cada robot (se inicializa al máximo).

- *self.zones* -> Contiene toda la información del mapa en forma de matriz, donde los valores de cada casilla siguen la siguiente asignación:

    0: Suelo limpio

    1: Suelo sucio

    2: Obstáculo

    3: Casilla de vaciado

    4: Punto de carga de robot rojo

    5: Punto de carga de robot verde

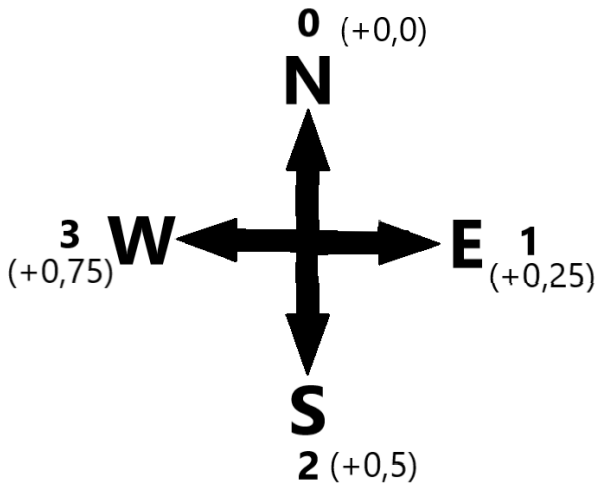
    6: Robot rojo

    7: Robot verde

- *self.ground* -> Guarda el valor de la casilla que está pisando el robot.

- *self.path* -> Variable para almacenar el camino recorrido por cada robot.

Además, para indicar la orientación de cada robot seguimos el siguiente esquema:



Explicándolo con un ejemplo: Si el robot rojo (‘6’ en la matriz *self.zones*) está orientado hacia el sur, significa que su valor en la matriz *self.zones* será ‘6,5’

Escogimos esta notación ya que nos facilitaba la implementación de la orientación dentro de la matriz del mapa.

A la hora de trabajar con las orientaciones, utilizaremos el formato con números enteros (mostrado anteriormente N=0, E=1, S=2, W=3), para ello debemos realizar los siguientes pasos:

1- Obtener el valor de orientación del robot (por ejemplo, el rojo): 6.75
2- Quedarnos solamente con la parte decimal: 0.75
3- Dividir el resultado por 0.25 para obtener el valor entero: $\frac{0.75}{0.25} = 3$ -> **Robot rojo orientado hacia W**

3. OPERADORES, PRECONDICIONES Y EFECTOS

Nuestros **operadores** serán los siguientes:

- Avanzar: El robot se desplaza una casilla

- Girar Izq: El robot gira en sentido antihorario 90º

- Girar Der: El robot gira en sentido horario 90º

Cuyas **precondiciones** para cada uno serán:

- ***Avanzar****(C,D,M)* ----------|-> C != 0; D != 0; M(robot1)[2] == 0; M[i, j-1] != (2||6||7) *

|-> C != 0; D != 0; M(robot1)[2] == 1; M[i+1, j] != (2||6||7)

|-> C != 0; D != 0; M(robot1)[2] == 2; M[i, j+1] != (2||6||7)

|-> C != 0; D != 0; M(robot1)[2] == 3; M[i-1, j] != (2||6||7)

Pseudo-código del operador *Avanzar*:

Si: tiene carga; tiene depósito libre; está orientado hacia (N=0,E=1,S=2,W=3); la casilla a la que está orientado no es ni un obstáculo ni el otro robot

Los **efectos** para cada una de las precondiciones anteriores del operador *Avanzar* serán:

*->|-> M[i, j-1] == 0; C = C-1; M[i, j] = 0

|-> M[i, j-1] == 1; C = C-1; D = D-1; M[i, j] = 0

|-> M[i, j-1] == 3; C = C-1; $D = D_{máx}$; M[i, j] = 3

|-> M[i, j-1] == 4; C = C-1 || $C = C_{máx}$; M[i, j] = 4 (Si el robot es el verde || Si el robot es el rojo)

|-> M[i, j-1] == 5; C = C-1 || $C = C_{máx}$; M[i, j] = 5 (Si el robot es el rojo || Si el robot es el verde)

(Solo se escriben los efectos para la primera condición para evitar explicaciones excesivas, las demás precondiciones tendrían los mismos efectos, pero solamente cambiaría el índice de la matriz de estado 'M')

Explicación de los efectos del operador *Avanzar:*

Si la casilla a la que se dirige es 0, 1, 3, 4 o 5, se actualizan los parámetros de carga y depósito correspondientes según la casilla y se actualiza la casilla de la que se vino el robot si corresponde

- ***GirarIzq****(C,D,M)* ----------|-> C != 0; M(robot1)[2] == 0; M[i-1, j] != (2||6||7) *

|-> C != 0; M(robot1)[2] == 1; M[i, j-1] != (2||6||7) *

|-> C != 0; M(robot1)[2] == 2; M[i+1, j] != (2||6||7) *

|-> C != 0; M(robot1)[2] == 3; M[i, j+1] != (2||6||7) *

Pseudo-código del operador *GirarIzq:*

Si: tiene carga; está orientado hacia (N=0,E=1,S=2,W=3); la casilla hacia la que se va a orientar no es ni un obstáculo ni el otro robot

Los **efectos** para todas las precondiciones anteriores del operador *GirarIzq* serán:

*-> C = C-0.5; M(6||7)[2] = *self.angles*[M(6||7)[2]-1]

Explicación de los efectos del operador *GirarIzq:*

Se disminuye el valor de la carga en 0.5 y se actualiza la orientación del robot correspondiente, esto lo realizamos obteniendo su valor de orientación (0,1,2,3) y obtenemos el valor anterior en la lista *self.angles = [0,1,2,3]* mediante indexación, que equivale a un giro en sentido antihorario.

- ***GirarDer****(C,D,M)* ----------|-> C != 0; M(robot1)[2] == 0; M[i+1, j] != (2||6||7)

|-> C != 0; M(robot1)[2] == 1; M[i, j+1] != (2||6||7)

|-> C != 0; M(robot1)[2] == 2; M[i-1, j] != (2||6||7)

|-> C != 0; M(robot1)[2] == 3; M[i, j-1] != (2||6||7)

Pseudo-código del operador *GirarDer:*

Si: tiene carga; está orientado hacia (N=0,E=1,S=2,W=3); la casilla hacia la que se va a orientar no es ni un obstáculo ni el otro robot

Los **efectos** para todas las precondiciones anteriores del operador *GirarDer* serán:

*-> C = C-0.5; M(6||7)[2] = *self.angles*[M(6||7)[2]+1]

Explicación de los efectos del operador *GirarDer:*

Se disminuye el valor de la carga en 0.5 y se actualiza la orientación del robot correspondiente, esto lo realizamos obteniendo su valor de orientación (0,1,2,3) y obtenemos el valor posterior en la lista *self.angles = [0,1,2,3]* mediante indexación, que equivale a un giro en sentido horario.

4. IMPLEMENTACIÓN

Teniendo toda nuestra lógica pensada y escrita procederemos a hacer la implementación de esta misma a un código de Python. Para ello dividiremos cada parte de lo pensado en partes del código.

La primera importante será el bloque central que hará que A* funcione, la función executeSearch(0 es para sin heurística y 1 es con heuristica). Esta la vamos a hacer con la ayuda de un diccionario donde iremos apuntando nuestro coste de cada nodo y *pq*, esta será una variable con la información del nodo inicial y donde irán todos los nodos. Tras esto empezaremos un bucle hasta que encuentre el estado final que buscamos (todo limpio y cada robot en su punto de carga). Para escoger el nodo con menor peso nos vamos a ayudar de una librería de Python llamada heapq, con ella tendremos una función (heappop) con la que podremos extraer de la lista de nodos el que tenga el menor costo de todos. Después de esto ira nuestra condición objetivo y tras ella un for que será el encargado de generar un nodo para cada acción que pueda hacer nuestros robots gracias a la función get_successors().

Esta función es la encargada de generar los nodos posteriores al nodo con menor valor. Dentro de ella para el cálculo de estas se ayudara de getValidActions() otra función que será la encargada de en el estado del nodo actual hallar las acciones posibles de cada robot, es decir, a partir de las precondiciones saber dónde se puede mover el robot. Tras esto comenzara a generar los nodos con las opciones que puede hacer el robot1, esto lo hará con apply_action() que es la función encargada de aplicar los efectos antes dichos. Cuando genere una acción del robot1 le pasara el nodo hecho al robot2 y hará lo mismo, primero mirara con getValidActions() las acciones posibles y después las aplicara con apply_action(). Es decir, primero se calcula el movimiento que va a hacer el robot1 y después el que va a hacer el robot2 con el movimiento del robot1 hecho. Esto lo hacemos asi para ahorrarnos problemas cara a que se choquen entre ello por hacer una acción que los coloque en la misma casilla, con esta solución nos libraremos de todos esos problemas perdiendo un poco de eficiencia, ya que el movimiento del robot2 siempre va a depender del movimiento del robot1.

Después que calcule todos los sucesores empieza a calcular el costo de que va a tener cada nodo. Los costos de cada nodo vienen dados por 3 variables (g, h, f).

- G: esta variable calcula el coste de la batería por hacer una acción, es decir, si por ejemplo el robot se mueve para delante este tendrá un coste de 1 y se le sumará a la g que ya haya en ese nodo.

- H: es el coste de la heurística. Se calcula en base a una formula desarrollada por nosotros.

- F: será el peso final. Este se calcula con la suma de G + H

Con esto hecho guardaremos el nodo siempre y cuando no exista en nuestra lista con los costes o que F sea inferior a un nodo igual ya existente. Si una de las dos se cumple se guardarán todos los nuevos y comenzara el bucle de nuevo con el nodo con menor coste. Asi, hasta que se encuentre la solución o nos quedemos sin nodos en *pq.*

5. HEURÍSTICA

La heurística en los problemas de A* nos ayudara a resolver más rápido y eficiente nuestro algoritmo. Con esta lo que buscaremos es añadirles peso extra a los nodos generados dependiendo del mapa y de las acciones que hagan los robots. Para este problema podemos dividir la heurística:

- Lo primero que haremos será mirar las zonas sucias del mapa, para ello haremos un bucle for del mapa y cada vez que salga una celda haremos la distancia manhattan de los dos robots hacia ese punto, la menor de estas será sumado a H

- Lo siguiente pasos se van a hacer individualmente para cada robot:

    - Haremos una suma de penalizaciones para castigar a los nodos donde el robot tenga un umbral de batería (menos de 4) o de suciedad baja (menos de 2).

    - Tras eso penalizaremos si el robot hace un giro siempre y cuando haga un giro cuando el umbral de batería baja (menos de 5) sea menor a la batería actual.

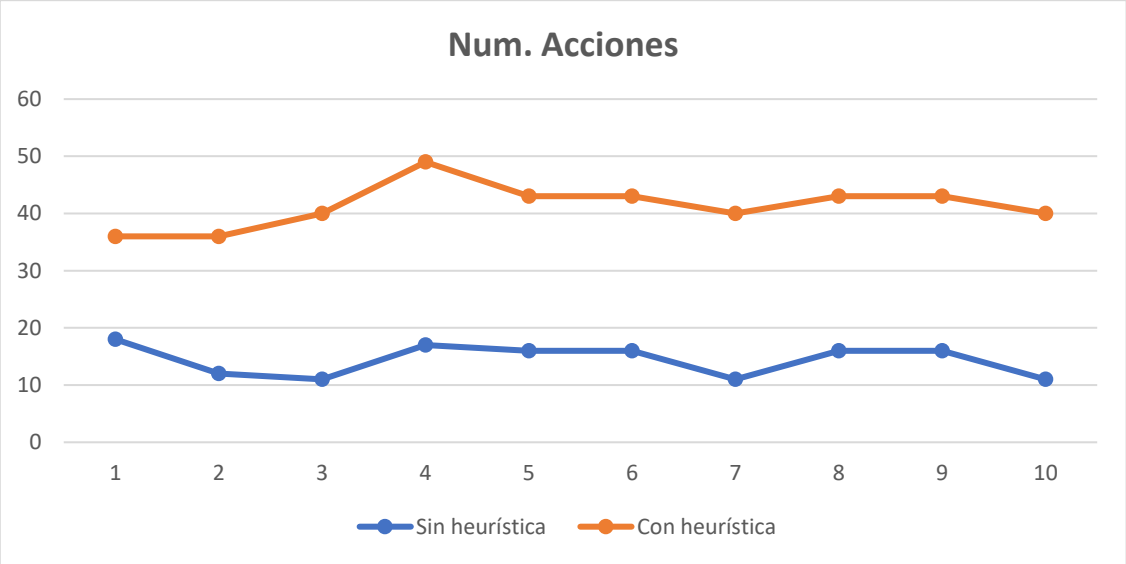
    - Por último, daremos peso a los nodos que cuando la batería sea menor al umbral de batería el robot este más cerca de su cargador.

Con esto ya tendríamos toda nuestra heurística hecha.

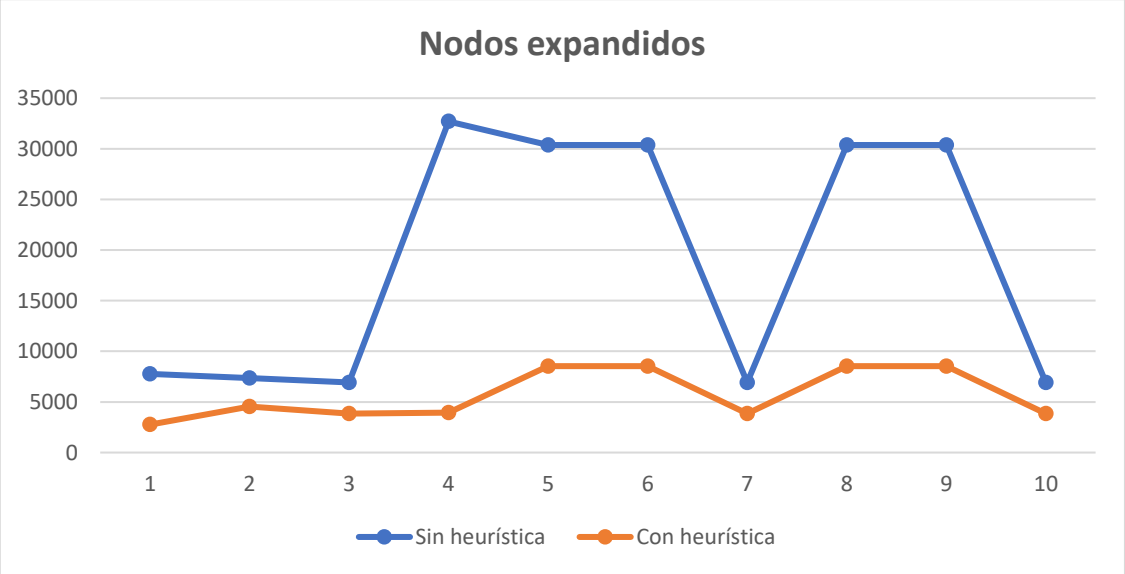
6. ANÁLISIS DE RESULTADOS

Ejecutando varias veces el algoritmo implementado para solucionar el problema de limpiar la habitación de tamaño 4x4 con localización de los obstáculos de manera aleatorias se han obtenido las siguientes medidas para la heurística 0 (Dijkstra) y para la desarrollada por nosotros:

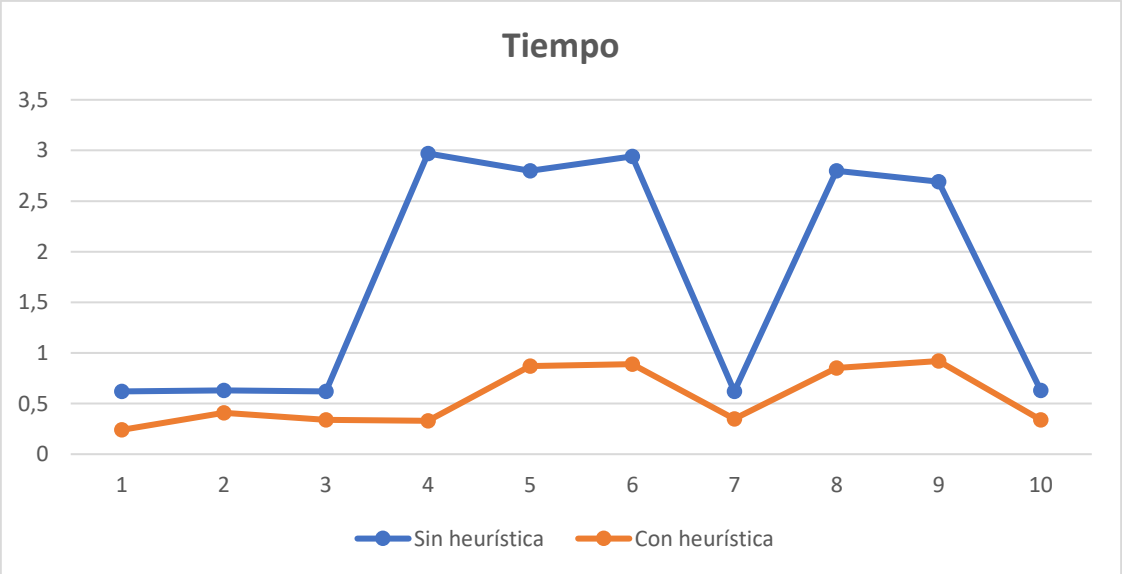
Acciones	Sin heurística	Con heurística
1	18	18
2	12	24
3	11	29
4	17	32
5	16	27
6	16	27
7	11	29
8	16	27
9	16	27
10	11	29
Media	13,88041237	26,31103644



Nodos expandidos	Sin heurística	Con heurística
1	7781	2759
2	7359	4554
3	6923	3858
4	32699	3954
5	30363	8534
6	30363	8534
7	6923	3858
8	30363	8534
9	30363	8534
10	6923	3858
Media	11627,01974	4804,77215



Tiempo	Sin heurística	Con heurística
1	0,62	0,24
2	0,63	0,41
3	0,62	0,34
4	2,97	0,33
5	2,8	0,87
6	2,94	0,89
7	0,62	0,35
8	2,8	0,85
9	2,69	0,92
10	0,63	0,34
Media	1,022894196	0,436453699



Observando las gráficas podemos obtener las siguientes conclusiones:

- **Estabilidad de los datos:** En el número de nodos generados y el tiempo de ejecución del algoritmo se puede ver un gran cambio a la hora de usar una heurística, ya que en algunos casos los resultados son similares, pero en otros puede llegar a ser mucho más costoso.
- **Optimización:** En general, la aplicación de nuestra heurística supone un costo menor en todos los campos del problema menos en el número de acciones requeridas para ejecutar un plan con solución, esto se debe a que nuestra heurística pretende que los robots no bajen de un cierto umbral de nivel de carga/espacio de depósito para así anticiparse a expandir nodos sin solución.

También intentamos aumentar el tamaño de la habitación a un mapa 5x5, pero el tiempo aumentó demasiado (t. heurística Dijkstra = 53 min., t. heurística propia = 23 min.) por lo tanto no tomamos medidas en mapas de mayor tamaño. La pregunta es, ¿por qué pasa esto, si nuestra heurística resuelve rápido los mapas de 4x4? El motivo de esta variación tan desorbitada en los tiempos obtenidos son los valores usados en la propia heurística. Usamos unos valores que calculamos más o menos haciendo pruebas en estos mapas y que resultan ser los mejores para nuestra heurística, pero para poder escalarla al siguiente nivel tendríamos que generalizar el cálculo de las variables que actúan dentro de la heurística como el umbral de batería o el coste de que gire.

7. CONCLUSIÓN

En esta memoria se ha detallado exhaustivamente la resolución del problema de limpieza de una habitación utilizando el algoritmo A*. Se planteó el desafío de manejar dos robots limpiadores que debían recorrer una habitación con varios elementos, como obstáculos, puntos de carga y áreas de vaciado, con el objetivo de limpiar de manera eficiente.

Se definieron los estados iniciales y los operadores, detallando las precondiciones y efectos de cada acción posible para los robots. Se optó por una representación matricial del entorno y se incorporó la orientación de los robots para facilitar la implementación.

La implementación del algoritmo A* se dividió en secciones, priorizando la generación de nodos sucesores a partir de un nodo dado y evaluando su costo, considerando la batería restante y el depósito de suciedad de los robots. Se desarrollaron funciones para obtener acciones válidas, aplicarlas en los estados y calcular el costo de cada nodo.

Se evaluó la eficacia de la implementación mediante métricas como el número de acciones realizadas, nodos expandidos y tiempo de ejecución. Se observó que la aplicación de una heurística propia supuso una mejora significativa en términos de nodos expandidos y tiempo de ejecución en comparación con el enfoque sin heurística. Sin embargo, hubo casos en los que el número de acciones requeridas para la solución fue mayor con la heurística, debido a la anticipación de las restricciones de carga y depósito.

Se intentó escalar la solución a habitaciones de mayor tamaño, pero se encontró un aumento considerable en el tiempo de ejecución. Se identificó que la variación en los tiempos se debía a la necesidad de generalizar los parámetros heurísticos para adaptarse a habitaciones más grandes.

En futuras iteraciones, la optimización y generalización de la heurística podrían ser clave para resolver eficientemente habitaciones de mayor tamaño, garantizando la escalabilidad y el rendimiento del algoritmo.