Curso:

Modelación de sistemas multiagentes con gráficas computacionales Gpo 301

Grupo:

301

Estudiante:

Paolo Zesati Negrete - A01784391

Maestro:

Profesor Octavio

Título:

Actividad: Roomba

Fecha de entrega:

18 de noviembre 2025

1. Problema que se está resolviendo y propuesta de solución

En esta práctica se modela el problema de limpieza autónoma de una habitación rectangular de tamaño M×N, en la que:

- Algunas celdas están inicialmente sucias.
- Algunas celdas actúan como obstáculos.
- Existen una o varias estaciones de carga para los roomba.
- Hay un tiempo máximo de ejecución.

El objetivo es diseñar agentes tipo “Roomba” capaces de **limpiar la mayor cantidad posible de celdas** dentro del tiempo máximo, **sin quedarse sin batería**, gestionando de forma inteligente su recarga.

1.2 Propuesta de solución

La solución se basa en un modelo multiagente implementado en MESA, con:

Un agente móvil RandomAgent (Roomba) con:

- Batería limitada (100% inicial).
- Costo energético por acción (1% por mover o limpiar).
- Capacidad de recarga en estaciones (5% por paso en una estación).
- Comportamiento reactivo basado en prioridades.
- Movimiento inteligente mediante un patrón tipo “snake” (lawnmower) que recorre su zona de forma sistemática.
- Priorización de celdas sucias vecinas y un contador de visitas para evitar repetir celdas.

Agentes fijos para representar:

- **FloorAgent:** celdas de piso (sucio/limpio).
- **ObstacleAgent:** obstáculos del modelo.
- **StationAgent:** estaciones de carga.

Se realizaron dos tipos de simulaciones:

Simulación 1 – Agente individual

- Un solo Roomba.
- Habitación con obstáculos y celdas sucias.
- El agente empieza en la celda [1,1], que también contiene una estación de carga.
- El agente recorre toda el área usando el patrón snake.

Simulación 2 – Múltiples agentes

- Varios Roombas.
- El grid se divide automáticamente en zonas verticales (una por agente).
- Cada agente inicia en una celda aleatoria dentro de su zona, donde también se coloca su estación de carga.
- Cada agente conoce la posición de su estación de origen, pero puede recargar en cualquier estación.
- Cada agente limpia exclusivamente dentro de su propia zona.

2. Diseño de los agentes

2.1 Objetivo del agente

El RandomAgent tiene como objetivo:

- Maximizar el número de celdas limpias durante el tiempo de simulación.
- Evitar quedarse sin batería, regresando a una estación de carga cuando su nivel sea bajo (30%).
- En términos de desempeño, busca un equilibrio entre:
    - Cobertura del área (explorar y limpiar).
    - Supervivencia (gestión de la batería).

2.2 Capacidades efectoras

El agente puede:

- Moverse a celdas vecinas sin obstáculos.
- Limpiar la celda actual si contiene un FloorAgent sucio.
- Permanecer sobre una estación de carga para recargar batería.
- Recorrer su zona mediante un patrón snake (move_snake_pattern).
- Moverse hacia celdas sucias vecinas (move_to_dirty_neighbor).
- Seleccionar el vecino con menos visitas (move_to_unvisited_neighbor).

En código, las acciones principales se implementan en:

- move_snake_pattern()
- move_to_dirty_neighbor()
- move_to_unvisited_neighbor()
- move_towards_home_station()
- clean_current_cell()
- charge_battery()

Cada acción de movimiento o limpieza consume 1% de batería.

2.3 Percepción del agente

El agente percibe:

El contenido de su celda actual:

- Si hay un FloorAgent y si está limpio/sucio.
- Si hay una StationAgent.
- Si hay obstáculos.

Las celdas vecinas (self.cell.neighborhood), para decidir:

- Si puede moverse (evita obstáculos).

- Si existe una celda sucia cercana.
- Cuál vecino ha sido visitado menos veces.

Contador de visitas

Mantiene un diccionario:

visit_count[(x,y)] para evitar pasar repetidamente por las mismas celdas.

La posición de su estación de carga “hogar”:

A través de home_station_pos.

2.4 Proactividad y comportamiento

El comportamiento del agente se define en el método step() y sigue una jerarquía de prioridades:

1. Seguridad energética

- Si battery <= 0, el agente deja de actuar.

2. Recarga

Si está sobre una estación (is_on_charging_station()):

- Si battery < 100: charge_battery().
- Si la celda está sucia: clean_current_cell().
- Si nada de lo anterior: move_to_random_neighbor() para salir a trabajar.

3. Regreso a la estación

Sí battery < 30:

- Llama a move_towards_home_station(),
  eligiendo el vecino que reduce la distancia Manhattan y evitando obstáculos.

4. Limpieza local

Si está sobre un FloorAgent sucio:

- Llama a clean_current_cell().

5. Movimiento estructurado (baja prioridad)

En caso contrario:

- Llama a move_snake_pattern().

El patrón snake incluye:

- Intentar seguir columna arriba/abajo.
- Evitar obstáculos.
- Cambiar de columna al llegar al extremo.
- Buscar vecinos sucios antes de avanzar.
- Elegir la celda menos visitada como último recurso.

2.5 Métricas de desempeño por agente

Cada RandomAgent lleva:

- battery: nivel actual de batería.
- movements: número de movimientos realizados.
- visit_count: número de visitas por celda (para evitar repetición).

En el modelo (RandomModel) se agregan métodos para recolectar:

- Movimientos por agente:
  get_roomba_movements() y get_roomba_movements_by_index().
- Conteo de celdas sucias y limpias:
  count_dirty_tiles() y count_clean_tiles().

Esto se usa para los plots de limpieza y movimientos.

3. Arquitectura de subsunción

El comportamiento del agente sigue una arquitectura de subsunción en capas:

Capa 1 – Supervivencia energética (máxima prioridad)

- battery <= 0 → no hace nada.

Capa 2 – Recarga en estación

Si está en una StationAgent:

- Si batería < 100: recarga.
- Si hay suciedad: limpia.
- Si está al 100% y limpio: sale a explorar.

Capa 3 – Regreso a la estación (“homing”)

- Si batería < 30 → move_towards_home_station().

Capa 4 – Limpieza local

- Si hay suciedad → clean_current_cell().

Capa 5 – Exploración estructurada (menor prioridad)

- move_snake_pattern()
  que incluye:
  - búsqueda de vecinos sucios,
  - evitar obstáculos,
  - movimiento snake,
  - elegir menor visit_count.

Esta capa sustituye la “exploración aleatoria”: ahora la exploración es dirigida y sistemática.

4. Características del ambiente

El entorno se modela mediante un OrthogonalMooreGrid:

- Dimensiones: width × height.
- Sin toro: torus=False.

- Bordes con ObstacleAgent simulando paredes.

4.1 Inicialización del ambiente

A partir de:

- Número de agentes.
- Celdas inicialmente sucias.
- Obstáculos internos.
- Tiempo máximo de ejecución.

Se hace:

Creación de bordes

Se colocan obstáculos en todas las celdas del borde.

Colocación de estaciones y agentes

- **Simulación 1:**
    - Estación en (1,1).
    - Un Roomba en esa celda.
- **Simulación 2:**
    - El dividen aleatoriamente los agentes a través del grid
    - Para cada zona:
        - Se elige una celda disponible.
        - Se coloca una estación y un agente.

Celdas de piso sucio

- Selección aleatoria de celdas no ocupadas.

Obstáculos internos

- Selección aleatoria de celdas vacías.

4.2 Características del ambiente particular

1. **Accesible vs. inaccesible?**
    a. En Roomba, cada agente sólo conoce su entorno inmediato (las celdas vecinas) y su posición de su estación de recarga, no todo el estado del mundo. No saben, por ejemplo, cuántas celdas faltan por limpiar o si hay otras Roombas. Es un ambiente inaccesible.
2. **Determinista vs. no determinista?**
    a. Aunque las reglas del modelo son fijas, el resultado de las acciones no siempre es predecible: El movimiento de exploración es aleatorio y en simulaciones con múltiples agentes, las posiciones libres pueden cambiar porque otros Roombas también se mueven. El orden de ejecución también varía (“shuffle_do”). Es un ambiente no determinista.
3. **Episódico vs. no episódico?**
    a. Las decisiones del agente dependen de lo que pasó antes: cuánta batería le queda, si ya limpió cierta celda, si ya regresó a cargar, etc. El estado actual del mundo afecta las acciones futuras. Es un ambiente no episódico.
4. **Estático vs. dinámico?**
    a. El ambiente cambia mientras el agente piensa/actúa: otros Roombas limpian, se mueven y ocupan celdas. Incluso con un solo agente, el tiempo afecta la batería y las celdas cambian de sucio→limpio. Es un ambiente dinámico.
5. **Discreto vs. continuo?**
    a. La habitación está dividida en celdas separadas y el tiempo avanza paso por paso; todas las acciones suceden en pasos discretos. Los agentes sólo pueden estar en una celda a la vez. Es un ambiente discreto.

5. Estadísticas recolectadas en las simulaciones

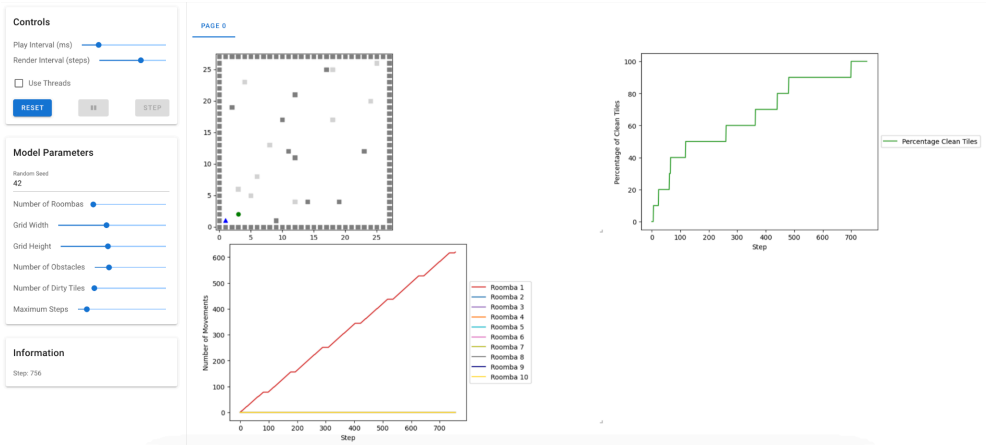
5.1 Simulación 1

En la simulación 1, el agente comienza en la celda [1,1], donde también se encuentra su StationAgent. Debido al consumo energético producido en cada acción (1% por movimiento o limpieza), la autonomía del Roomba se ve limitada, obligándo a regresar a su estación cuando la batería cae por debajo del 30%. Para este experimento realicé tres simulaciones, variando únicamente la cantidad de celdas sucias iniciales: 10, 15 y 20.
El objetivo es analizar si un mayor número de *dirty tiles* influye en el tiempo necesario para limpiar toda la habitación.

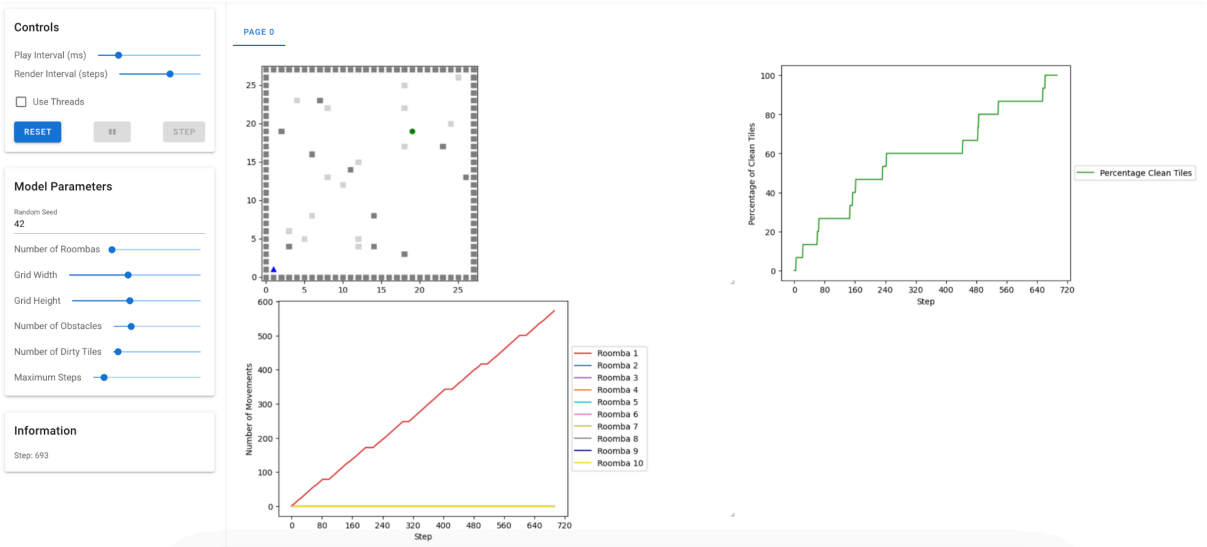
Parámetros estáticos del modelo

- Número de Roombas: 1
- Ancho del grid: 28
- Alto del grid: 28
- Número de obstáculos: 10

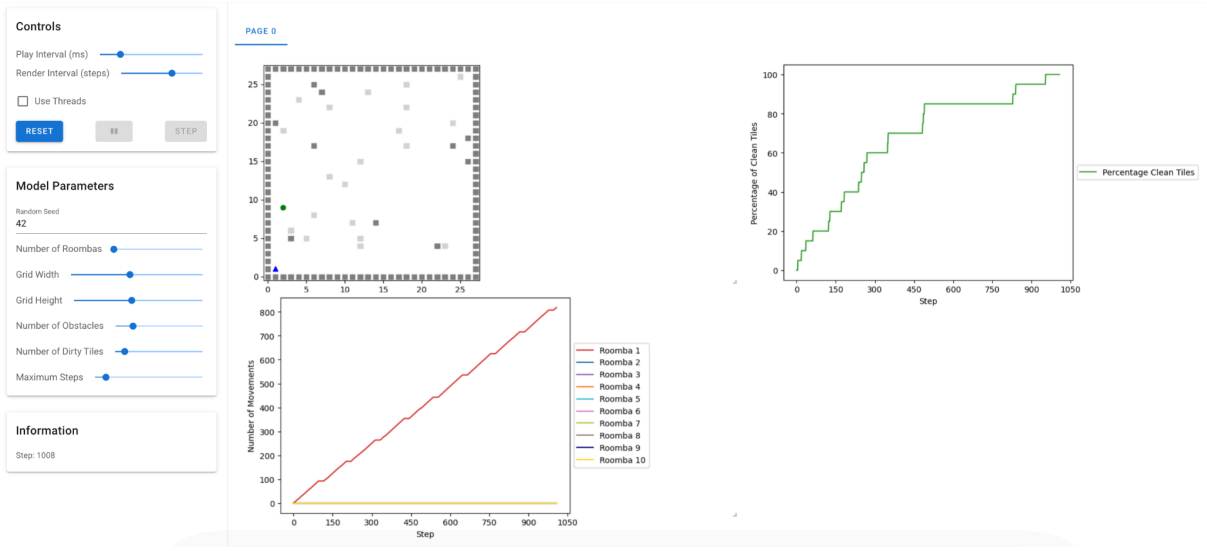
5.1.1 (Número de Dirty Tiles: 10)



5.1.2 (Número de Dirty Tiles: 15)



5.1.3 (Número de Dirty Tiles: 20)



5.1.4 Conclusión

En estas pruebas observé que incrementar ligeramente el número de celdas sucias no necesariamente incrementa el número de pasos de manera proporcional. Entre las simulaciones con 10 y 15 celdas sucias, el tiempo total fue prácticamente el mismo.

Sin embargo, con 20 dirty tiles sí hubo un aumento notable en los pasos necesarios para completar la limpieza. Aun así, considero que todas las simulaciones fueron exitosas, ya que el agente logró completar su tarea dentro de los límites establecidos.

En pruebas adicionales detecté que el factor que más incrementa el número de pasos no es el número de celdas sucias, sino la cantidad de obstáculos. Con más obstáculos, el patrón de movimiento snake se interrumpe con mayor frecuencia, obligando al agente a recalcular rutas y visitar más celdas repetidas. Por ello, una posible mejora para el modelo es implementar un mecanismo más avanzado de evasión de obstáculos, lo cual podría reducir significativamente el número de pasos necesarios.

5.2 Simulación 2

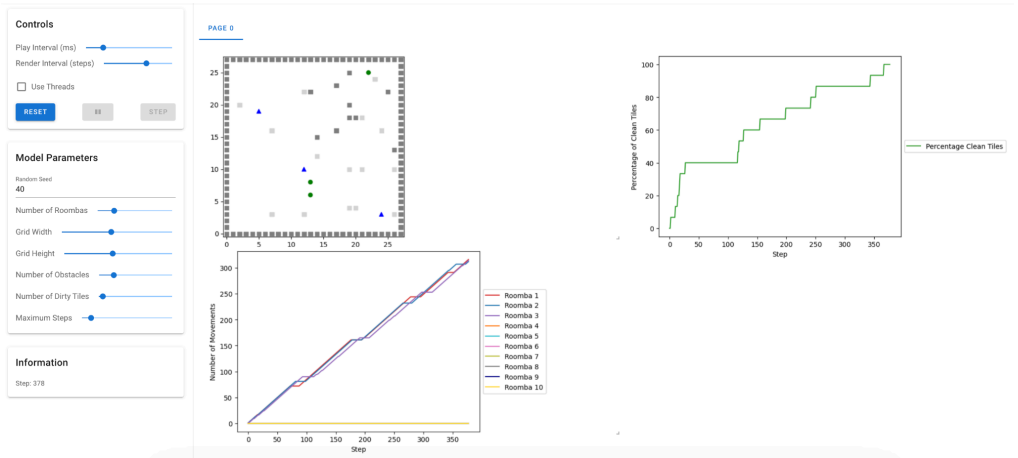
Tras analizar el comportamiento del agente individual en la Simulación 1, donde evaluamos cómo la cantidad de celdas sucias afecta el desempeño del Roomba bajo restricciones de batería y movimiento, ahora extendemos el experimento hacia un escenario multiagente. El objetivo de esta segunda parte es observar cómo cambia la eficiencia del modelo cuando varios Roombas operan simultáneamente dentro del mismo entorno, cada uno asignado a una zona específica del grid. Esto permite evaluar si la limpieza se acelera al aumentar el número de agentes y cuáles son las limitaciones emergentes cuando no existe comunicación ni coordinación entre ellos.

Parámetros estáticos del modelo

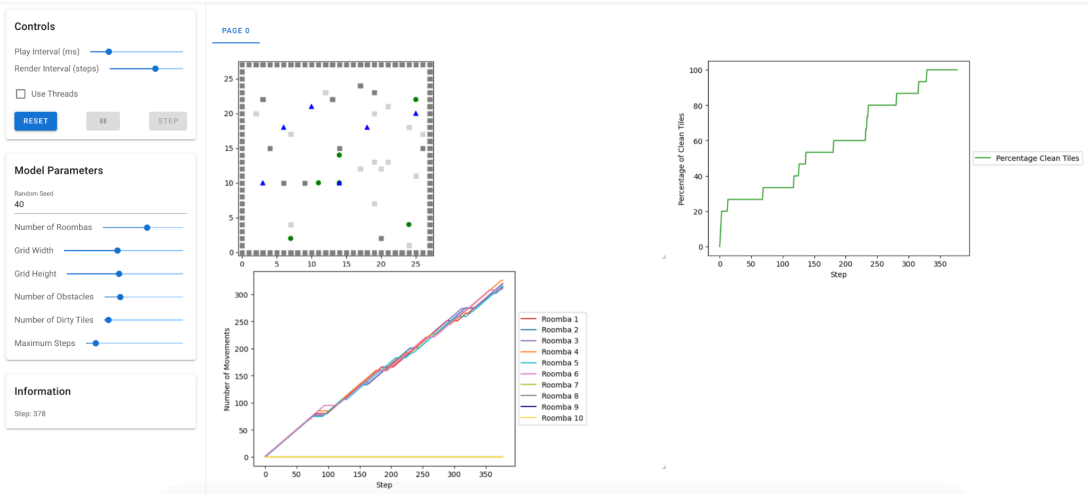
- Ancho del grid: 28
- Alto del grid: 28
- Número de obstáculos: 10
- Número de dirty tiles: 15
- Máximo de pasos: 2000

En esta simulación evaluamos el efecto de añadir múltiples Roombas, dividiendo el grid en zonas verticales asignadas automáticamente.

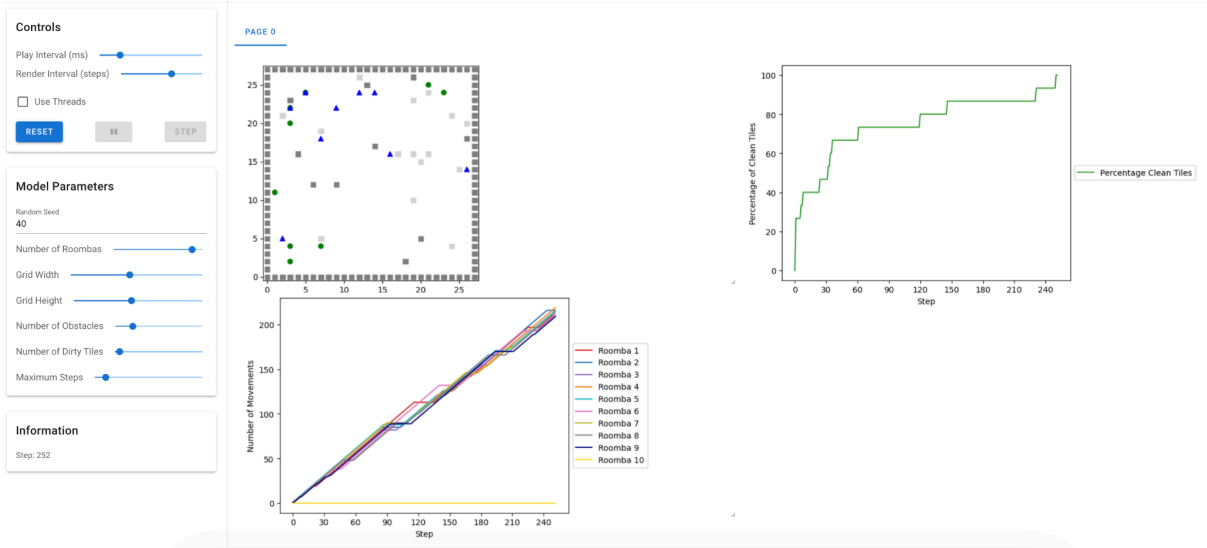
5.2.1 (Number of Roombas 3)



5.2.2 (Number of Roombas 6)



5.2.3 (Number of Roombas 9)



5.2.4 Conclusión

Los resultados muestran que añadir más Roombas reduce el número total de pasos necesarios para completar la limpieza. Sin embargo, la mejora no es lineal:

- Entre 3 y 6 agentes casi no hubo diferencia significativa.
- Entre 6 y 9 agentes, en cambio, sí se observó una reducción considerable en el tiempo de limpieza.

Esto sugiere que el modelo se beneficia de la cooperación implícita entre agentes, pero también revela una limitación importante: actualmente los Roombas no se comunican entre sí.

Cada uno opera únicamente dentro de su zona y sin compartir información como:

- Celdas ya visitadas
- Celdas limpias
- Rutas óptimas

Si en futuras versiones se implementará un sistema de comunicación o compartición de mapas, los agentes podrían:

- Evitar visitar celdas ya recorridas por otros

- Reducir duplicación de movimiento
- Minimizar el tiempo total de simulación

Aún sin comunicación, este ejercicio permitió observar claramente los beneficios y límites de escalar la cantidad de agentes en un ambiente multiagente.

Conclusiones

A lo largo de esta actividad se logró modelar de manera efectiva un sistema multiagente tipo Roomba capaz de limpiar un entorno discreto bajo restricciones energéticas, espaciales y temporales. En la Simulación 1, se observó que un solo agente puede completar la tarea aun con un consumo energético constante, aunque su eficiencia depende más de la distribución y cantidad de obstáculos que del número de celdas sucias iniciales. Esto evidencia la importancia del diseño de rutas y de mejores mecanismos de evasión de obstáculos para reducir pasos innecesarios. En la Simulación 2, el análisis mostró que incorporar múltiples agentes sí acelera el proceso de limpieza, aunque la mejora no es lineal: el rendimiento solo aumenta de forma significativa a partir de cierto número de Roombas. Esta limitación se debe principalmente a la ausencia de comunicación entre agentes y a que cada uno opera únicamente con percepción local, lo que puede generar solapamientos en las rutas y redundancia en los movimientos.

En conjunto, los resultados permiten concluir que el modelo es funcional y cumple con los objetivos planteados, pero también señalan áreas claras de mejora, como integrar estrategias cooperativas, compartir información entre agentes o implementar algoritmos de planificación más sofisticados. Estas extensiones permitirían aumentar la eficiencia global del sistema y explorar comportamientos más complejos y realistas dentro de un entorno multiagente.