Planeación de Movimiento de Robótica Multi-Agente

Jose Quintana
Métodos y Modelos Computacionales
Universidad Tecnológica de Pereira
Pereira, Colombia
jose.quintana@utp.edu.co

***Resumen*—El presente informe detalla la planificación de movimiento en robótica multi-agente para un espacio de trabajo rectangular, en el cual se distribuyen obstáculos y tareas que se mueven aleatoriamente. El objetivo principal consiste en desarrollar un algoritmo que optimice el movimiento de los agentes para minimizar la distancia total recorrida, teniendo en cuenta las prioridades de las tareas cuando estas existen.**

Se presenta una descripción de la metodología empleada para la solución de la actividad, explicando la formulación del problema de optimización para minimizar la distancia total recorrida por los agentes, los algoritmos utilizados para calcular las trayectorias más cortas y la asignación de tareas. A continuación, se muestran los resultados obtenidos en diversos casos de prueba para validar la efectividad del programa. Se concluye con un análisis que permite comparar el rendimiento y la eficiencia de las estrategias propuestas, y se realizan observaciones de las aplicaciones de los sistemas descritos en la vida real.

***Palabras clave*—Sistemas Robóticos, Planificación del Movimiento, Algoritmo de Optimización, Programación Lineal, Camino más Corto.**

***Abstract*—This report details the motion planning in multi-agent robotics for a rectangular workspace, in which randomly moving obstacles and tasks are distributed. The main objective is to develop an algorithm that optimizes the movement of the agents to minimize the total distance traveled, taking into account the priorities of the tasks when they exist.**

A description of the methodology used for the solution of the activity is presented, explaining the formulation of the optimization problem to minimize the total distance traveled by the agents, the algorithms used to calculate the shortest paths and the assignment of tasks. The results obtained in various test cases are then shown to validate the effectiveness of the program. It concludes with an analysis to compare the performance and efficiency of the proposed strategies, and observations are made on the real-life applications of the described systems.

***Keywords*—Robotics Systems, Motion Planning, Optimization Algorithm, Linear Programming, Pair Shortest Path.**

I. Introducción

La robótica ha recorrido un largo camino desde sus inicios hasta convertirse en una de las tecnologías más influyentes y avanzadas de nuestra era: desde la automatización de tareas industriales hasta la exploración espacial, los robots han demostrado ser herramientas esenciales que mejoran la eficiencia y la precisión en una gran variedad de aplicaciones [1]. Los avances de los últimos años en materia de Industrias 4.0 han posibilitado la creación de robots más autónomos y adaptables [2]. En este contexto, la planificación de movimiento en robótica tiene un papel fundamental para optimizar estas capacidades, asegurando que los robots puedan desplazarse y operar de manera eficiente y segura en entornos complejos y dinámicos.

En relación a la planificación de robots, se distinguen dos enfoques principales: sistemas mono-agente y multi-agente. La planificación de movimiento para los primeros se centra en la secuencia óptima de acciones que un solo robot debe realizar para completar una tarea específica; por otro lado, la planificación de movimiento para los segundos implica la coordinación y cooperación de múltiples robots que trabajan en conjunto para alcanzar objetivos comunes. Este último enfoque es particularmente trascendental en aplicaciones donde la tarea es demasiado grande o complicada para un solo robot, o donde la cooperación entre varios robots puede resultar en un rendimiento significativamente mejorado.

Sin embargo, la planificación de movimiento en sistemas de robótica multi-agente presenta múltiples desafíos, ya que la coordinación efectiva entre los robots es crucial para evitar colisiones y garantizar que las tareas se completen de manera eficiente. Esto requiere algoritmos sofisticados capaces de gestionar las trayectorias de múltiples robots simultáneamente, teniendo en cuenta las restricciones del entorno y los posibles conflictos. Además, la minimización de colisiones y la optimización del tiempo y los recursos son aspectos críticos que deben abordarse para garantizar un funcionamiento fluido y seguro del sistema. En este sentido, la integración de distintas herramientas matemáticas y computacionales resulta fundamental para proponer una solución adecuada a problemáticas.

En este contexto, la teoría de grafos, los algoritmos avaros y la programación dinámica son herramientas esenciales en la planificación de movimientos de robots [3]. La teoría de grafos proporciona un marco para modelar las relaciones espaciales y de movimiento entre los nodos (posiciones) y aristas (caminos) en el entorno de los robots. Mediante la implementación de algoritmos avaros, que buscan soluciones óptimas locales en cada paso, se realiza una búsqueda eficiente y rápida de rutas para dicho entorno. La programación dinámica, por su parte, divide los problemas complejos en subproblemas más simples y resuelve cada uno de manera óptima, permitiendo una solución global eficiente. En conjunto, estas técnicas se

complementan para desarrollar algoritmos de planificación robustos y efectivos en escenarios de robótica multi-agente donde la coordinación y la optimización son fundamentales.

En el presente informe se plantean dos actividades principales. La primera consiste en la generación de un espacio de trabajo, conformado por celdas cuadradas. En dicho espacio, se distribuyen obstáculos y tareas, representados por celdas cerradas y círculos rojos respectivamente, que se mueven aleatoriamente en intervalos inciertos. Además, se disponen agentes (representados por tortugas) que deben alcanzar las tareas según el caso: siguiendo un orden de prioridad si las tareas lo tienen, o minimizando las distancias recorridas en ausencia de prioridades. La segunda actividad consiste en conseguir que los agentes alcancen las tareas mencionadas, según los criterios del caso.

La metodología para resolver las actividades planteadas incluye la implementación del algoritmo de Floyd-Warshall para calcular las distancias más cortas entre los nodos y el uso de programación lineal para realizar la asignación de tareas a los agentes. Se formula un problema de optimización que minimiza la distancia total recorrida por los agentes, y se establecen restricciones para asegurar una asignación adecuada de las tareas. Por último, se presentan los resultados obtenidos en diversos casos de prueba etiquetados para validar la efectividad del programa.

II. Desarrollo

Para abordar la implementación de la planificación de movimiento en robótica multi-agente en Python, se utilizó la aplicación "Labyrinth Project" [4], que permite crear una representación gráfica de un espacio de trabajo rectangular dividido en pequeñas celdas cuadradas haciendo uso de la librería Tkinter [5]. Adicionalmente, se empleó un enfoque de multi-threading con la librería "threading" [6], asegurando que tanto la ventana del lienzo del espacio de trabajo como la planificación de movimiento puedan ejecutarse de manera concurrente.

La metodología empleada en este trabajo evidencia un enfoque sistemático y estructurado para la resolución de problemas complejos en el ámbito de la robótica multi-agente. Para ello, se sigue un proceso claro que comienza con la generación del entorno y los obstáculos, en conjunto con la implementación de algoritmos de búsqueda de caminos y la optimización de tareas. La descomposición del problema muestra cómo un desafío complejo puede ser abordado de manera efectiva dividiéndolo en subproblemas como la creación del espacio de trabajo y la planificación de movimientos.

II-A. Espacio de trabajo

Se crean dos procesos principales, que se ejecutan a través de la implementación de hilos: uno para la función `create_labyrinth` que se encarga de crear la ventana del espacio de trabajo, y otro para la función `solve_problem`, la cual plantea el algoritmo de generación tanto del espacio de trabajo con obstáculos, tareas y agentes de manera aleatoria, y también la planificación de movimiento de los agentes para alcanzar dichas tareas.

II-A1. Ventana Emergente

La función `create_labyrinth` crea un objeto Labyrinth con el número especificado de filas y columnas, e inicia el bucle de eventos de Tkinter para mostrar el espacio de trabajo rectangular dividido en pequeñas celdas cuadradas en la pantalla, como se puede ver en Fig 5.

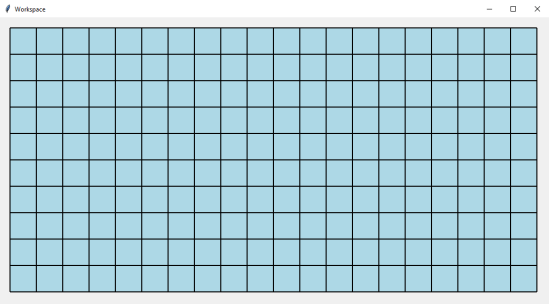

Figura 1. Espacio de Trabajo Rectangular

La clase 'Labyrinth' incluye métodos como inicializar el espacio de trabajo (`start`), crear el espacio de trabajo (`create_maze`), calcular el tamaño de las celdas (`cell_size`), actualizar el espacio de trabajo según el respectivo grafo (`update_maze`), obtener el mosaico específico de la lista list_tiles (`get_tile`) y marcar la posición y dirección de la tortuga en el espacio de trabajo (`mark_turtle`), entre otros. La implementación de la función es la siguiente:

```
def create_labyrinth():
    maze = Labyrinth(ROWS, COLUMNS)
    maze.start()
```

II-A2. Grafo con Obstáculos

Para abordar el desafío de la generación de obstáculos (celdas cerradas) en el espacio de trabajo rectangular, se propone la generación de componentes conexas [7] . Una componente conexa es un conjunto de nodos que están conectados entre sí, y no están conectados con otros nodos fuera de la componente. Para cualquier par de nodos en la componente, hay un camino entre ellos conformado únicamente por nodos de la componente.

La función `generar_celda_cerrada()` se encarga de crear estas componentes conexas. Esta función utiliza funciones auxiliares como `es_valido()` y `obtener_nodo()` para verificar la validez de las celdas y obtener el nodo correspondiente a una celda. Dichas funciones auxiliares son las siguientes:

```
def es_valido(x, y, rows, columns):
    return 0 <= x < rows and 0 <= y < columns
```

La función `es_valido` toma cuatro argumentos: `x` y `y`, que representan las coordenadas de una celda en el espacio de trabajo, y `rows` y `columns`, que representan el número total de filas y columnas del espacio de trabajo, respectivamente. La función verifica si las coordenadas `x` e `y` están dentro de los límites del espacio de trabajo. Devuelve `True` si `x` está

entre 0 y el número total de filas menos uno, y `y` está entre 0 y el número total de columnas menos uno; de lo contrario, devuelve `False`.

```
def obtener_nodo(x, y, columns):
    return x * columns + y
```

La función `obtener_nodo` toma tres argumentos: `x` y `y`, que representan las coordenadas de una celda en el espacio de trabajo, y `columns`, que representa el número total de columnas del espacio de trabajo. Esta convierte las coordenadas de la celda en un índice único de nodo basado en un sistema de numeración lineal. El índice del nodo se calcula multiplicando `x` por el número total de columnas y sumando `y`, lo que proporciona una identificación única para cada celda en el espacio de trabajo en función de sus coordenadas.

La generación de componentes conexas (celdas cerradas) surgen a partir de la función `generar_celda_cerrada()`, como se observa a continuación:

```
def generar_celda_cerrada(rows, columns, n):
    componente_conexa = set()

    start_x, start_y = random.randint(0, rows
    ↪  - 1), random.randint(0, columns - 1)
    componente_conexa.add(obtener_nodo(
        start_x, start_y, columns))

    while len(componente_conexa) < n:
        nodo_actual = random.choice(list(
            componente_conexa))
        x, y = nodo_actual // columns,
        ↪  nodo_actual % columns
        vecinos = [(x + dx, y + dy) for dx, dy
        ↪  in [(-1, 0), (1, 0), (0, -1), (0,
        ↪  1), (-1, -1), (-1, 1), (1, -1),
        ↪  (1, 1)] if es_valido(x + dx, y +
        ↪  dy, rows, columns)]
        if vecinos:
            vecino_x, vecino_y =
            ↪  random.choice(vecinos)
            nodo_vecino =
            ↪  obtener_nodo(vecino_x,
            ↪  vecino_y, columns)
            componente_conexa.add(nodo_vecino)

    return componente_conexa
```

En la Fig. 2 se observa una celda cerrada, conformada por nodos de componentes conexas:

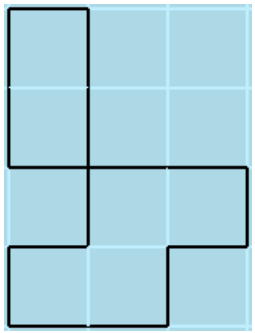

Figura 2. Celdas Cerradas en Espacio de Trabajo

Estos serán los obstáculos que se podrán visualizar en el espacio de trabajo final.

Posteriormente, se genera un grafo con obstáculos a partir de estas componentes conexas utilizando la función `generar_grafo_con_obstaculos(rows, columns, m)`, donde `rows` y `columns` son el número de filas y columnas del espacio de trabajo rectangular, respectivamente, y `m` es el número de componentes conexas que se desean generar. Se añaden aristas entre nodos adyacentes que no estén en la componente conexa para crear el perímetro de celdas cerradas, y se añaden aristas entre nodos de componentes conexas para permitir el movimiento de los agentes entre estas componentes.

```
def generar_grafo_con_obstaculos(rows,
↪  columns, m):
    grafo = Grafo()

    obstaculos = []
    for _ in range(m):
        n = random.randint(1, 6)
        print("Números de nodos que componen
        ↪  la componente conexa: ", n)
        componente_conexa =
        ↪  generar_celda_cerrada(rows,
        ↪  columns, n)
        obstaculos.append(componente_conexa)
```

Primero, la función `generar_grafo_con_obstaculos` inicializa un objeto `grafo` y una lista `obstaculos`. Luego, genera `m` componentes conexas aleatorias de tamaños entre 1 y 6 nodos, utilizando la función `generar_celda_cerrada`, y las almacena en la lista `obstaculos`.

```
    for x in range(rows):
        for y in range(columns):
            nodo = obtener_nodo(x, y, columns)
            if any(nodo in celda for celda in
            ↪  obstaculos):
                continue

            for dx, dy in [(-1, 0), (1, 0),
            ↪  (0, -1), (0, 1), (-1, -1),
            ↪  (-1, 1), (1, -1), (1, 1)]:
                nx, ny = x + dx, y + dy
                if es_valido(nx, ny, rows,
                ↪  columns) and
                ↪  all(obtener_nodo(nx, ny,
                ↪  columns) not in celda for
                ↪  celda in obstaculos):
                    nodo_vecino =
                    ↪  obtener_nodo(nx, ny,
                    ↪  columns)
                    # Definir diagonales
                    if dx != 0 and dy != 0:
                        grafo.add_edge(nodo,
                        ↪  nodo_vecino,
                        ↪  math.sqrt(2))
                    else:
                        grafo.add_edge(nodo,
                        ↪  nodo_vecino, 1)
```

A continuación, la función itera sobre cada celda del espacio de trabajo representado por las filas `rows` y columnas `columns`. Para cada celda, se verifica si es parte de una componente conexa en la lista `obstaculos`. Si no lo es, se evalúan sus celdas adyacentes (incluyendo diagonales) y, en

caso de que sean válidas y no pertenezcan a una componente conexa, se añaden aristas al grafo con un peso de 1 para aristas ortogonales y $\sqrt{2}$ para aristas diagonales.

```
for celda in obstaculos:
    for nodo in celda:
        x, y = nodo // columns, nodo %
        ↪  columns
        for dx, dy in [(-1, 0), (1, 0),
        ↪  (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if es_valido(nx, ny, rows,
            ↪  columns):
                nodo_vecino =
                ↪  obtener_nodo(nx, ny,
                ↪  columns)
                if nodo_vecino in celda:
                    grafo.add_edge(nodo,
                    ↪  nodo_vecino, 1)
                else:
                    grafo.add_edge(nodo,
                    ↪  nodo_vecino, 0)

        for dx, dy in [(-1, 0), (1, 0),
        ↪  (0, -1), (0, 1), (-1, -1),
        ↪  (-1, 1), (1, -1), (1, 1)]:
            nx, ny = x + dx, y + dy
            if es_valido(nx, ny, rows,
            ↪  columns):
                nodo_vecino =
                ↪  obtener_nodo(nx, ny,
                ↪  columns)
                if nodo_vecino in celda:
                    grafo.add_edge(nodo,
                    ↪  nodo_vecino,
                    ↪  math.sqrt(2))
```

Para cada nodo en las componentes conexas, se añaden aristas entre nodos adyacentes dentro de la misma componente para permitir el movimiento entre ellos. Las aristas ortogonales tienen un peso de 1 y las diagonales tienen un peso de $\sqrt{2}$. Si los nodos adyacentes no pertenecen a la misma componente, se añade una arista con peso 0, indicando un perímetro de celda cerrada.

```
nodos_obstaculos = [nodo for celda in
↪  obstaculos for nodo in celda]
print("Nodos de las componentes conexas:
↪  ", nodos_obstaculos)

return grafo, nodos_obstaculos
```

Finalmente, se crea una lista `nodos_obstaculos` que contiene todos los nodos pertenecientes a las componentes conexas y se imprime para referencia. La función retorna el `grafo` generado y la lista `nodos_obstaculos`. Dichos obstáculos se pueden visualizar en Fig. 3.

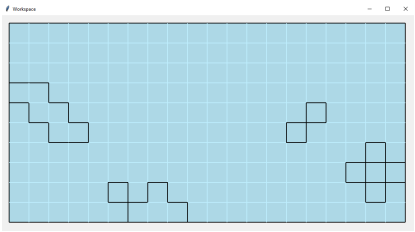

Figura 3. Espacio de Trabajo con Obstáculos

II-A3. Tareas y Agentes

Se generan tareas y agentes de forma aleatoria en el espacio de trabajo, evitando los nodos de las componentes conexas (celdas cerradas) como posiciones de tareas y agentes. Las tareas se representan como círculos rojos, y los agentes como tortugas en el grafo. La asignación de colores y posiciones se realiza de la siguiente manera:

```
grafo.colors = {nodo_tarea: 'red' for
↪  nodo_tarea in tareas}
grafo.turtle = {nodo_agente: nodo_agente - 1
↪  for nodo_agente in agentes}
```

El espacio de trabajo resultante con los obstáculos, tareas y agentes distribuidos de forma aleatoria se puede apreciar en la Fig. 4.

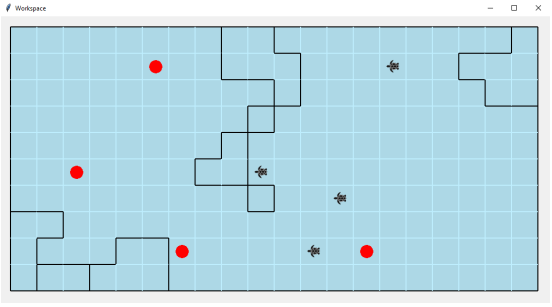

Figura 4. Espacio de Trabajo con Obstáculos, Tareas y Agente

II-B. Camino más corto entre todos los pares de nodos

El desafío consiste en determinar los caminos más cortos entre cada par de nodos en un grafo dado. Esto se presenta en términos de un grafo ponderado $G = (V, E)$, donde se busca encontrar, para cada par de nodos (u, v) en el conjunto V de nodos, el camino de menor peso posible entre u y v. En este contexto, el peso de un camino se define como la suma de los pesos de las aristas que lo componen [7].

II-B1. Algoritmo de Floyd-Warshall

Una formulación de programación lineal para encontrar el problema del camino más corto de todos los pares en un grafo ponderado $G = (V, E)$ es el algoritmo Floyd-Warshall [7].

El algoritmo de Floyd-Warshall considera los nodos intermedios de un camino más corto, lo cual disminuye la complejidad del problema, donde un nodo intermedio de un camino simple $p = \{v_1, v_2, \ldots, v_l\}$ es cualquier nodo de p distinto de v_1 o v_l, es decir, cualquier nodo en el conjunto de $\{v_2, v_3, \ldots, v_{l-1}\}$.

La implementación de dicho algoritmo en pseudocódigo se observa a continuación:

Algorithm 1 Algoritmo Floyd-Warshall (W)

$n = W.\text{rows}$
$D^{(0)} = W$
for $k = 1$ to n **do**
  let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
  for $i = 1$ to n **do**
    for $j = 1$ to n **do**
      $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
    end for
  end for
end for
return $D^{(n)}$

En este contexto, el algoritmo Floyd-Warshall se utiliza para calcular las distancias entre las celdas de un espacio de trabajo, considerando obstáculos. Se presta especial atención a las aristas diagonales, las cuales se invalidan si los nodos intermedios pertenecen a componentes conexas, dado que los agentes no pueden atravesar celdas cerradas. Esto se implementa asegurando que las aristas diagonales que conectan nodos de la misma componente conexa no se consideren en el cálculo de las distancias más cortas, como se ilustra en la Figura 5.

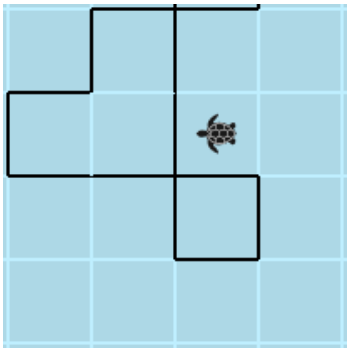

Figura 5. Arista Diagonal Invalida.

El algoritmo Floyd-Warshall con obstáculos se implementa de la siguiente manera:

Inicialización

```
def FloydWarshall(grafo, nodos_obstaculos):
    V = grafo.V
    E = grafo.E
    inf = float('inf')
    dist = {u: {v: inf for v in V} for u in V}
    next_node = {u: {v: None for v in V} for u
    ↪  in V}
```

Inicialmente, se obtienen los conjuntos de nodos (`V`) y aristas (`E`) del grafo. Se define `inf` como infinito y se inicializan las matrices de distancias (`dist`) y de nodos siguientes (`next_node`) con valores apropiados.

Asignación de Distancias Iniciales

```
    for u in V:
        dist[u][u] = 0
        for v in V[u]:
            if f"({u}, {v})" in E and not
            ↪  es_arista_diagonal_invalida(u,
            ↪  v, nodos_obstaculos):
                dist[u][v] = E[f"({u}, {v})"]
                next_node[u][v] = v
            if f"({v}, {u})" in E and not
            ↪  es_arista_diagonal_invalida(v,
            ↪  u, nodos_obstaculos):
                dist[v][u] = E[f"({v}, {u})"]
                next_node[v][u] = u
            if dist[u][v] != dist[v][u]:
                dist[u][v] = dist[v][u] =
                ↪  min(dist[u][v],
                ↪  dist[v][u])
                next_node[u][v] = v
                next_node[v][u] = u
```

Para cada nodo u, se establece la distancia a sí mismo como 0. Luego, para cada vecino v de u, se actualizan las distancias y los nodos siguientes siempre y cuando la arista no sea una arista diagonal inválida.

Actualización de Distancias

```
    for k in V:
        if k in nodos_obstaculos:
            continue
        for i in V:
            if i in nodos_obstaculos:
                continue
            for j in V:
                if j in nodos_obstaculos:
                    continue
                if dist[i][j] > dist[i][k] +
                ↪  dist[k][j]:
                    dist[i][j] = dist[i][k] +
                    ↪  dist[k][j]
                    next_node[i][j] =
                    ↪  next_node[i][k]
    return dist, next_node
```

En esta parte, se actualizan las distancias mínimas entre cada par de nodos i y j, considerando a k como nodo intermedio, solo si i, j y k no son nodos obstáculos. Si se encuentra una distancia más corta a través de k, se actualiza la matriz de distancias y la matriz de nodos siguientes.

II-C. Planificación de Movimiento

La planificación de movimientos de los agentes para alcanzar las tareas asignadas (implementada en la función `planificar_movimientos`) se realiza en dos escenarios:

1. Cuando las tareas tienen prioridad, se asignan las tareas a los agentes en orden de prioridad decreciente y se planifican las rutas óptimas para los agentes. El orden de prioridad de las tareas se define aleatoriamente asignando un número entre 1 y 10 a cada tarea, donde un número mayor indica una mayor prioridad.

```
orden_prioridad = {t:
↪  random.randint(1, 10) for t in
↪  tareas}
```

2. Cuando las tareas no tienen prioridad, se selecciona la tarea más cercana al agente, minimizando la distancia total recorrida por el agente, y se planifican las rutas óptimas para los agentes.

II-C1. Minimización de distancia total

La programación lineal (PL) es una técnica matemática de optimización que busca maximizar o minimizar una función objetivo lineal, sujeta a un conjunto de restricciones lineales. El 'Método Simplex', desarrollado por George Dantzig en 1947, es un algoritmo eficaz para resolver problemas de programación lineal [8]. Este algoritmo itera sobre los vértices de la región factible para encontrar la solución óptima del problema PL.

En un problema estándar de programación lineal, se busca optimizar una función objetivo lineal sujeta a un conjunto de restricciones lineales, y puede ser formulado como:

Maximización:

$$\begin{aligned} &\text{Maximizar} && c^T x \\ &\text{sujeto a} && Ax \leq b \\ & && x \geq 0 \end{aligned}$$

Minimización:

$$\begin{aligned} &\text{Minimizar} && c^T x \\ &\text{sujeto a} && Ax \geq b \\ & && x \geq 0 \end{aligned}$$

donde x es el vector de variables, c es el vector de coeficientes de la función objetivo, A es la matriz de coeficientes de las restricciones y b es el vector de términos independientes.

El Método Simplex inicia en un vértice de la región factible y se mueve a lo largo de los vértices adyacentes, mejorando el valor de la función objetivo en cada paso hasta alcanzar la solución óptima. Su implementación en pseudocódigo está dada así:

Algorithm 2 Algoritmo Simplex para Programación Lineal

1: **Inicialización:** Seleccionar una solución básica factible inicial $x^{(0)}$
2: **Iteración:**
3: **repeat**
4: Calcular los costos reducidos y determinar la dirección de búsqueda
5: Determinar la variable básica que abandonará la base y la variable no básica que entrará en la base
6: Actualizar la solución básica factible según la dirección de búsqueda
7: **until** Todos los costos reducidos son no negativos
8: **Terminación:** El algoritmo termina cuando todos los costos reducidos son no negativos, alcanzando así la solución óptima

En el problema de asignación de tareas, la programación lineal y el Método Simplex se utilizan para encontrar la distribución óptima de tareas a los agentes, minimizando la distancia total recorrida y asegurando que se cumplan todas las restricciones.

La optimización de la asignación de tareas a agentes minimizando la distancia total recorrida por los estos se realiza mediante programación lineal entera (ILP) con la librería PuLP [9]. Se utiliza el solver CBC para resolver el problema de asignación de tareas a agentes [10]. Se define un problema de minimización donde las variables de decisión son binarias y representan si una tarea se asigna o no a un agente. Las restricciones se establecen para asegurar que cada tarea se asigne a un agente y que cada agente se asigne al menos a una tarea, según sea conveniente.

La función objetivo de minimización y las restricciones se formulan matemáticamente como sigue:

Función Objetivo: Minimizar la distancia total recorrida por los agentes:

$$\text{Minimizar} \sum_{i=1}^{N} \sum_{j=1}^{M} d_{ij} x_{ij}$$

donde d_{ij} es la distancia entre el agente i y la tarea j, y x_{ij} es una variable binaria que indica si la tarea j se asigna al agente i.

- $x_{ij} = 1$ indica que la tarea j es asignada al agente i.
- $x_{ij} = 0$ indica que la tarea j no es asignada al agente i.
- N: Número total de agentes.
- M: Número total de tareas.

Restricciones:

1. Cada tarea debe ser asignada exactamente a un agente:

$$\sum_{i=1}^{N} x_{ij} = 1 \quad \forall j \in \{1, \ldots, M\}$$

2. Cada agente debe hacer al menos una tarea si hay más tareas que agentes, en cuyo caso puede ser asignado a más de una tarea:

$$\begin{cases} \sum_{j=1}^{M} x_{ij} \leq M & \text{si } M > N \\ \sum_{j=1}^{M} x_{ij} \leq 1 & \text{si } M \leq N \end{cases} \quad \forall i \in \{1, \ldots, N\}$$

La función `optimization` implementa un algoritmo para minimizar la distancia total recorrida por los agentes al asignar tareas. A continuación, se presenta el código de la función en secciones, junto con su explicación:

Inicialización del Problema

```
def optimization(agentes, tareas, dist):
    prob = pulp.LpProblem("AssignmentProblem",
    ↪  pulp.LpMinimize)
    num_agentes = len(agentes)
    num_tareas = len(tareas)
```

Se crea un problema de minimización y se obtienen los conjuntos de agentes y tareas.

Definición de Variables de Decisión

```
    x = pulp.LpVariable.dicts("x", ((i, j) for
    ↪  i in range(num_agentes) for j in
    ↪  range(num_tareas)), cat='Binary')
```

Se definen las variables de decisión binarias x_{ij}.

Función Objetivo

```
prob +=
↪ pulp.lpSum(dist[agentes[i]][tareas[j]]
↪ * x[i, j] for i in range(num_agentes)
↪ for j in range(num_tareas))
```

La función objetivo minimiza la distancia total recorrida.

Restricciones

- **Restricción 1**: Cada tarea debe ser asignada a un solo agente.

```
# Restricción #1:
for j in range(num_tareas):
    prob += pulp.lpSum(x[i, j] for i
    ↪ in range(num_agentes)) == 1
```

- **Restricción 2**: Cada agente puede ser asignado a más de una tarea, pero debe hacer al menos una tarea si hay más tareas que agentes

```
# Restricción #2:
if num_tareas > num_agentes:
    for i in range(num_agentes):
        prob += pulp.lpSum(x[i, j] for
        ↪ j in range(num_tareas)) <=
        ↪ num_tareas
else:
    for i in range(num_agentes):
        prob += pulp.lpSum(x[i, j] for
        ↪ j in range(num_tareas)) <=
        ↪ 1
```

Resolución del Problema

```
prob.solve(pulp.PULP_CBC_CMD())
```

El problema se resuelve utilizando el solucionador CBC.

Obtención de Asignaciones

```
assignments = [(i, j) for i in
↪ range(num_agentes) for j in
↪ range(num_tareas) if
↪ pulp.value(x[i, j]) == 1]

return assignments
```

Se obtienen las asignaciones de tareas a agentes.

II-C2. Reconstrucción de Rutas

La función `reconstruir_ruta` se implementa en la función principal de planificación de movimiento `planificar_movimientos`. La reconstrucción de la ruta entre dos nodos en el grafo se realiza utilizando los caminos más cortos calculados con el algoritmo de Floyd-Warshall. Se valida la ruta para evitar aristas diagonales inválidas, asegurando que los agentes no atraviesen celdas cerradas.

La función `reconstruir_ruta` toma tres parámetros:

- `next_node`: Una estructura que contiene los nodos siguientes en el camino más corto entre los nodos del grafo.
- `inicio`: El nodo de inicio de la ruta.
- `destino`: El nodo de destino de la ruta.

Validación inicial

```
def reconstruir_ruta(next_node, inicio,
↪ destino)
    if inicio is None or destino is None:
        return []
    if next_node[inicio][destino] is None and
    ↪ next_node[destino][inicio] is None:
        return []
```

Se realiza una verificación inicial para asegurarse de que tanto el nodo de inicio como el nodo de destino sean válidos. Si alguno de estos nodos es `None`, la función retorna una lista vacía, indicando que no existe una ruta válida entre los nodos proporcionados. Además, se verifica si no hay un camino directo (arista) entre el nodo de inicio y el nodo de destino en ambas direcciones. Si no hay una conexión directa, también se retorna una lista vacía.

Inicialización de la ruta

```
ruta = [inicio]
```

Para iniciar la reconstrucción de la ruta, se inicializa una lista `ruta` con el nodo de inicio. Esto establece el punto de partida desde el cual se empezará a reconstruir la ruta hacia el nodo de destino.

Reconstrucción de la ruta

```
while inicio != destino:
    siguiente = next_node[inicio][destino]
    if siguiente is None:
        break
    ruta.append(siguiente)
    inicio = siguiente
```

Se utiliza un bucle `while` para reconstruir paso a paso la ruta entre el nodo de inicio y el nodo de destino. El bucle continúa mientras el nodo actual de `inicio` no sea igual al nodo `destino`. En cada iteración, se consulta la estructura `next_node` para obtener el siguiente nodo en el camino más corto hacia el `destino` desde el `inicio` actual. Si no hay un siguiente nodo disponible (es decir, `siguiente` es `None`), se interrumpe el bucle. De lo contrario, se agrega el `siguiente` nodo a la lista `ruta` y se actualiza `inicio` para continuar la reconstrucción de la ruta.

Retorno de la ruta

```
return ruta
```

Finalmente, la función retorna la lista `ruta`, que contiene todos los nodos ordenados que forman la ruta reconstruida desde el `inicio` hasta el `destino`.

II-D. Movimiento de los Agentes

Se simulan los movimientos de los agentes en el espacio de trabajo rectangular. Cada agente se mueve hacia la tarea asignada siguiendo la ruta óptima. El grafo se actualiza continuamente con la posición de los agentes y el estado de las tareas, que cambian de color a verde una vez completadas, permitiendo una visualización clara del progreso en el espacio de trabajo.

```
def mover_agente(agente, ruta, grafo,
↪ candado, tareas):
    if not ruta:
```

```
            return
        for i in range(len(ruta) - 1):
            grafo.turtle[ruta[i]] = ruta[i +
            ↪  1]específica)
            with candado:
                grafo.send_graph()
            time.sleep(1/4)
            grafo.turtle.pop(ruta[i])
            if ruta[i + 1] in tareas:
                grafo.colors[ruta[i + 1]] =
                ↪  'green' específica)
                with candado:
                    grafo.send_graph()
        grafo.turtle[ruta[-1]] = ruta[-1]
        time.sleep(1/2)
        grafo.turtle.pop(ruta[-1])
        with candado:
            grafo.send_graph()
```

La función principal para resolver el problema de planificación de movimiento de los agentes es `solve_problem`, la cual se encarga de recopilar todas las funciones previamente explicadas y ejecutarlas en un ciclo infinito. Gracias a esto, se genera el espacio de trabajo rectangular con obstáculos, se añaden tareas y agentes aleatoriamente en el espacio de trabajo, se planifican los movimientos de los agentes para alcanzar las tareas asignadas, y se mueven los agentes en el espacio de trabajo.

La función `solve_problem` comienza con un bucle infinito, lo que indica que el proceso de generación de espacio de trabajo, asignación de tareas y movimiento de agentes se ejecuta continuamente.

```
def solve_problem():
    while True:
```

Primero, se genera un grafo con obstáculos, es decir, celdas cerradas o componentes conexas, explicado en la función `generar_grafo_con_obstaculos`. El número de componentes conexas `m` se elige aleatoriamente entre 1 y 6, definido de forma arbirtaria.

```
        m = random.randint(1, 6) # Número de
        ↪  componentes conexas (celdas
        ↪  cerradas)
        grafo, nodos_obstaculos =
        ↪  generar_grafo_con_obstaculos(ROWS,
        ↪  COLUMNS, m)
        print(grafo.get_graph())
```

Luego, se generan aleatoriamente `k` tareas y `n` agentes en el espacio de trabajo rectangular, evitando los nodos que forman parte de las componentes conexas. Los nodos válidos (aquellos que no son obstáculos) se almacenan en la lista `nodos_validos`, de la cual se seleccionan aleatoriamente `k` nodos para las tareas y `n` nodos para los agentes.

```
        k = random.randint(1, 6)
        n = random.randint(1, 4)

        nodos_validos = [nodo for nodo in
        ↪  range(ROWS * COLUMNS) if nodo not
        ↪  in nodos_obstaculos]

        tareas = random.sample(nodos_validos,
        ↪  k)
        agentes = random.sample(nodos_validos,
        ↪  n)
```

A continuación, se verifica que no haya superposición entre los nodos asignados a las tareas y los agentes. Si un nodo se asigna a ambos, se elimina de una de las listas. También se asegura que las tareas no se ubiquen en nodos de componentes conexas.

```
        for nodo_tarea in tareas:
            if nodo_tarea in agentes:
                agentes.remove(nodo_tarea)

        for nodo_agente in agentes:
            if nodo_agente in tareas:
                tareas.remove(nodo_agente)

        for nodo_tarea in tareas:
            if nodo_tarea in nodos_obstaculos:
                tareas.remove(nodo_tarea)
```

Las tareas y los agentes se añaden al grafo, donde las tareas se representan como círculos rojos y los agentes como tortugas. El grafo se actualiza y se envía a la cola para su visualización, con un breve retardo para permitir la actualización de la interfaz gráfica.

```
        grafo.colors = {nodo_tarea: 'red' for
        ↪  nodo_tarea in tareas}
        grafo.turtle = {nodo_agente:
        ↪  nodo_agente - 1 for nodo_agente in
        ↪  agentes}
        # Actualizar el grafo en la cola
        with candado:
            grafo.send_graph()
        time.sleep(1/8)
```

Se calculan las distancias más cortas entre los nodos utilizando la función de `FloydWarshall`. Además, se asigna un orden de prioridad aleatorio a las tareas, donde un número mayor indica una mayor prioridad.

```
        dist, next_node = FloydWarshall(grafo,
        ↪  nodos_obstaculos)

        orden_prioridad = {t:
        ↪  random.randint(1, 10) for t in
        ↪  tareas}
```

Se planifican los movimientos de los agentes para alcanzar las tareas asignadas usando la función `planificar_movimientos`. Luego, se crean hilos para mover cada agente según las rutas planificadas, iniciando cada hilo y esperando a que todos los hilos terminen antes de continuar con la siguiente iteración del ciclo infinito.

```
        rutas_totales =
        ↪  planificar_movimientos(agentes,
        ↪  tareas, dist, next_node,
        ↪  nodos_obstaculos,
        ↪  prioridad=PRIORIDAD,
        ↪  orden_prioridad=orden_prioridad)

        hilos = []
        for agente, ruta in
        ↪  rutas_totales.items():
            hilo = threading.Thread(
    target=mover_agente, args=(ruta, grafo,
    ↪  candado, tareas))
            hilos.append(hilo)
            hilo.start()

        for hilo in hilos:
            hilo.join()
```

Finalmente, se espera a que los hilos de `solve_problem` y `create_labyrinth` terminen, completando el proceso de generación del espacio de trabajo y la planificación de movimiento de los agentes.

III. Resultados

En esta sección se muestran los resultados obtenidos al ejecutar el algoritmo desarrollado para la planificación de movimiento de los agentes. En la Fig. 6, se muestra un espacio de trabajo de 10 filas y 20 columnas con un número aleatorio de obstáculos, tareas y agentes.

A modo de ejemplo, se realiza el análisis de la ejecución del algoritmo en la planificación de movimiento con un solo agente. Esto permite visualizar claramente la trayectoria de la tortuga para alcanzar cada una de estas tareas. Se contrastan dos casos: cuando las tareas tienen prioridad, basados en su peso asociado, o sin prioridad teniendo en cuenta las distancias.

Es fundamentar mencionar que, si bien para la explicación se utilizó un caso con un sistema mono-agente por cuestiones de facilidad de visualización, la planificación del movimiento aplicada funciona de la misma manera para sistemas multi-agente. En el repositorio [11] se encuentran vídeos que documentan la ejecución de esta metodología para una gran variedad de casos.

Al ejecutar el código, se muestran en la terminal las posiciones en las que se encuentran las tareas (círculos rojos) en el espacio de trabajo de la Fig. 6, así:

```
Tareas:  [120, 124, 22, 175]
```

Es necesario aclarar que la numeración empieza desde cero, para la celda ubicada en la esquina superior izquierda, y termina en $(NColumns * MRows)$ - 1 para la celda inferior derecha.

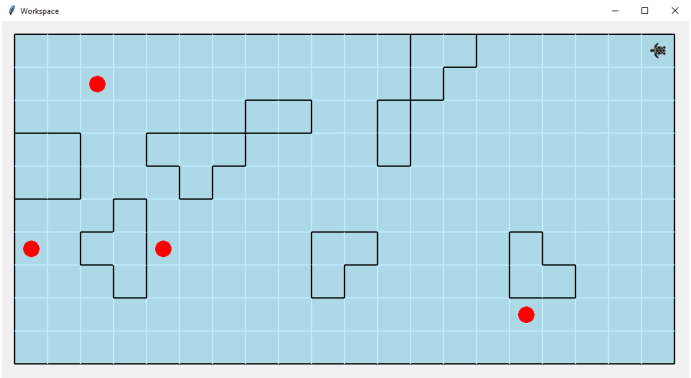

Figura 6. Espacio de Trabajo 10x20.

A continuación, se realiza un análisis de la implementación para casos con prioridad y sin ella. Es posible observar, en las figuras, las tareas asignadas al agente como círculo rojos. Sin embargo, en el momento en que el agente completa una tarea, esta se convierte en un círculo verde para indicar su nuevo estado.

III-A. Implementación con Prioridad

En la terminal, se indica la prioridad asociada a cada tarea. Esto se realiza considerando que en la función `planificar_movimientos` dichas tareas se ordenan de manera decreciente según su peso. Por tanto, entre mayor sea el valor asociado, mayor prioridad se establece para la tarea.

```
Orden de prioridad de las tareas:  {120: 5, 124: 2,
↪  22: 1, 175: 2}
```

En la Fig. 7, se alcanza en primera instancia la tarea en posición '120'. Esto es debido a que su peso (5) es el de mayor valor entre las tareas asignadas, como se explicó anteriormente.

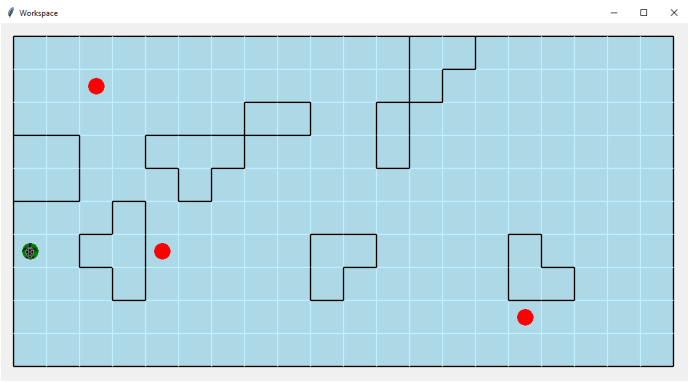

Figura 7. Primera Tarea Alcanzada (CP).

A continuación, el agente se dirige a la posición '124' como segunda tarea. Bajo la misma lógica anterior, su peso de valor '2' le concedió dicha prioridad. El cambio de posición del agente se observa en la Fig. 8.

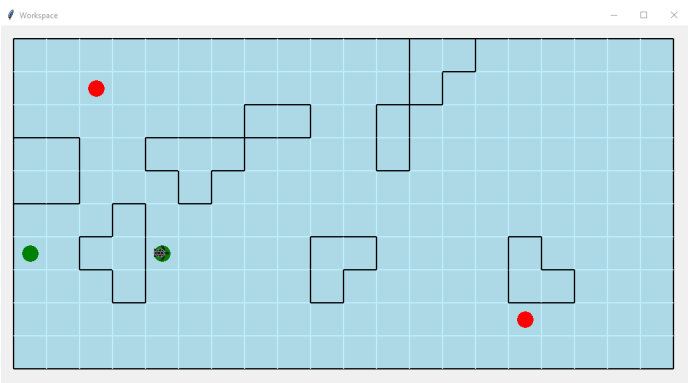

Figura 8. Segunda Tarea Alcanzada (CP).

En la Fig.9 se observa cómo el agente llega a la celda de posición '175' en tercer lugar, debido a su peso de '2' entre las tareas. Debido a que el peso de este objetivo es el mismo que el del apartado anterior, se determinó la prioridad de forma tal que se minimice la distancia total recorrida por la tortuga.

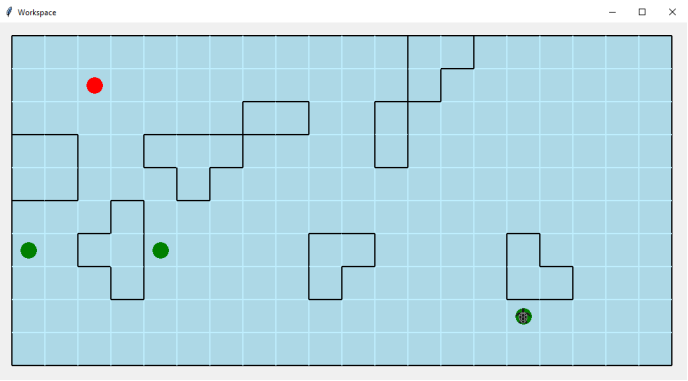

Figura 9. Tercera Tarea Alcanzada (CP).

Para la cuarta y última tarea, ubicada en la posición '22', su prioridad se estableció debido a que su peso '1' es el de menor valor entre las tareas generadas. Como se observa en la Fig. 10, el agente desaparece una vez completadas todas sus asignaciones.

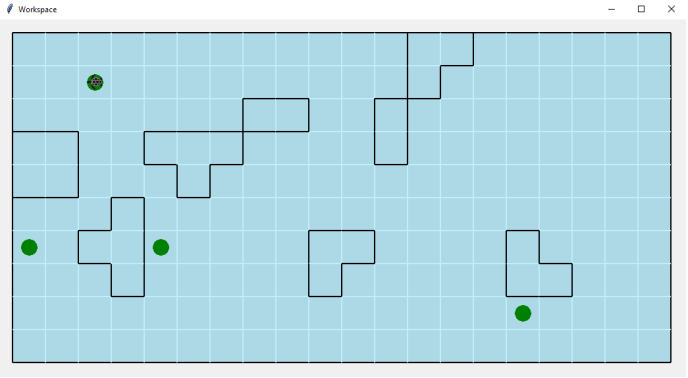

Figura 10. Todas las Tareas Alcanzadas (CP).

III-B. Implementación sin Prioridad

Para el segundo caso a analizar, no se asocia un peso a las tareas generadas. Por tanto, la trayectoria que recorre el agente para alcanzar las tareas se calcula con el objetivo de minimizar la distancia total recorrida.

La primera tarea alcanzada se observa en la Fig. 11. Este orden se asignó ya que es la posición más cercana con la aparición inicial del agente.

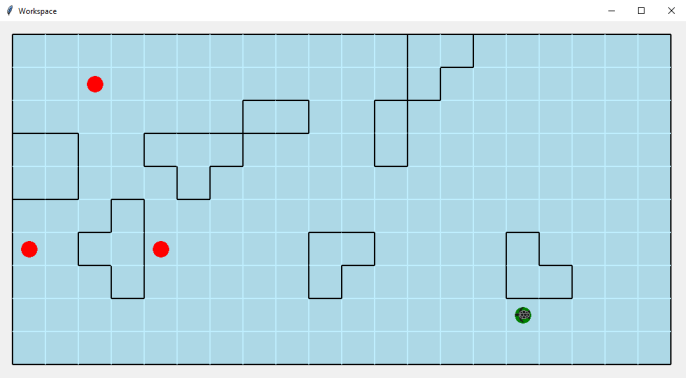

Figura 11. Primera Tarea Alcanzada (SP).

En la Fig.12 se muestra la tarea que alcanza el agente en segunda instancia:

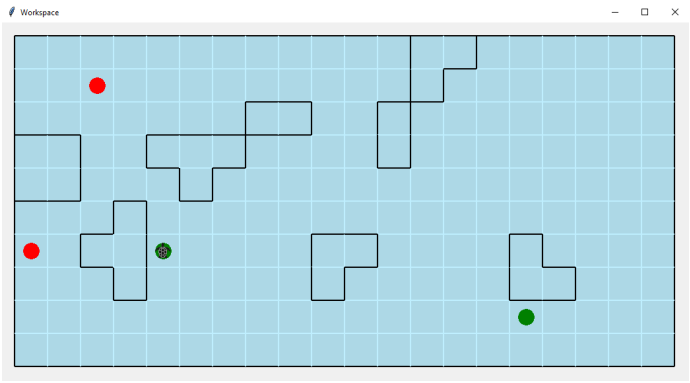

Figura 12. Segunda Tarea Alcanzada (SP).

La tercera tarea conseguida se puede visualizar en Fig. 13:

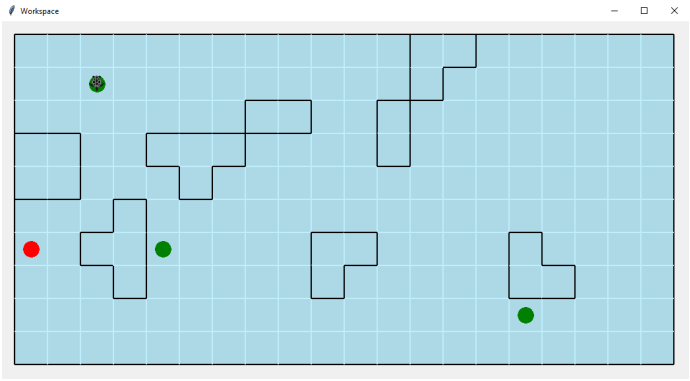

Figura 13. Tercera Tarea Alcanzada (SP).

Por último, se alcanza la tarea cuatro y se completa la planificación de movimiento de la tortuga, como se muestra en la Fig. 14:

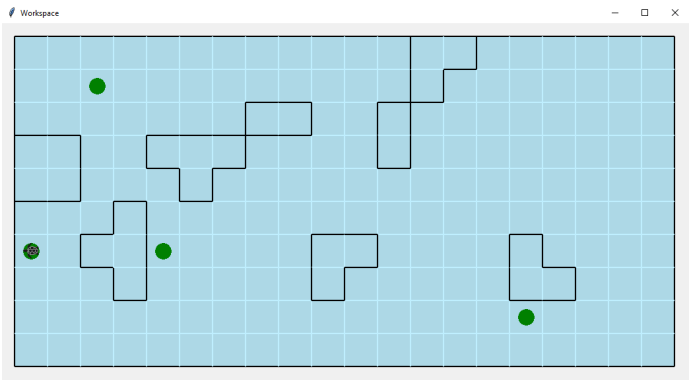

Figura 14. Todas las Tareas Alcanzadas (SP).

IV. Conclusión

Para la solución de la actividad planteada, se realizó una implementación computacional cuyo objetivo es realizar la planificación de movimiento de varios agentes a través de

un espacio de trabajo con obstáculos y tareas generados de forma aleatoria. Esta implementación se conforma de varios algoritmos, cada uno encargado de un elemento del sistema global.

El algoritmo central del proyecto es aquel encargado del cálculo de las trayectorias que debe realizar cada agente para alcanzar de forma óptima las tareas distribuidas en el espacio de trabajo. Este debe tener en cuenta dos casos: si las tareas cuentan con un peso asignado, debe existir una prioridad para resolverlas. Por otro lado, es posible que no se cuente con pesos, en lo cual se define el recorrido que deben realizar los agentes teniendo en cuenta la distancia total del mismo. Por tanto, es necesario abordar el asunto desde diferentes perspectivas.

Existen diversos algoritmos que permiten calcular el camino más corto entre todos los pares de nodos, modelando el espacio de trabajo como un grafo. Para la implementación explicada, se utilizó el algoritmo Floyd-Warshall ya que este considera nodos intermedios de un camino. Por tanto, su costo computacional es menor frente a otras alternativas. Empleando este algoritmo, se calcularon las rutas basadas en los caminos más cortos respetando el orden establecido de las tareas, si lo hay, para el desplazamiento de los agentes.

Adicionalmente, para lograr una adecuada implementación de estrategias de planificación de movimiento, se hace fundamental llevar a cabo la formulación matemática de los problemas de optimización involucrados en el sistema. Una formulación idónea se basa en definir claramente la función objetivo, y las restricciones necesarias asociadas a ella. De esta forma, al momento de presentar los resultados, se propicia la obtención de soluciones que estén alineadas con el propósito planteado.

Las estrategias aplicadas posibilitan la escalabilidad del sistema y los resultados obtenidos demuestran la efectividad del algoritmo en mejorar la eficiencia operativa de los agentes. En escenarios con alta densidad de tareas y obstáculos, se observa un rendimiento con capacidad para coordinar a los agentes, reduciendo la distancia total recorrida y evitando las barreras que se encuentran en el espacio de trabajo.

En implementaciones reales, existen diversas aplicaciones prácticas para sistemas similares al propuesto. Por ejemplo, en robótica industrial, la generación de obstáculos y componentes conexas es crucial para la navegación autónoma en entornos complejos, como almacenes automatizados. En estos ambientes, los robots deben evitar áreas restringidas (obstáculos) mientras se mueven entre estaciones de trabajo (componentes conexas) para recoger y entregar mercancías. Esta técnica garantiza que los robots eviten colisiones y optimicen sus rutas, mejorando la eficiencia operativa y reduciendo tiempos de inactividad.

Por otro lado, algoritmos basados en encontrar el camino más corto entre todos los pares de nodos, como el algoritmo de Floyd-Warshall, son fundamentales en redes de transporte y telecomunicaciones. En logística, este algoritmo permite determinar las rutas más eficientes entre múltiples centros de distribución, minimizando los costos de transporte. Otra aplicación incluye redes de telecomunicaciones, en donde ayuda a encontrar las rutas óptimas para la transmisión de datos, garantizando una comunicación rápida y eficiente entre nodos de la red.

Por último, es importante destacar que la planificación de movimiento en robótica multiagente, en general, se extiende a diversos campos como la exploración espacial, la gestión de emergencias y el mantenimiento de infraestructuras. En la exploración espacial, por ejemplo, múltiples robots pueden colaborar para mapear terrenos desconocidos, recolectar muestras y construir bases, optimizando sus movimientos para conservar recursos limitados. En la gestión de emergencias, equipos de drones pueden coordinarse para buscar y rescatar personas en áreas de desastre, mientras evitan obstáculos y cubren el máximo terreno posible de manera eficiente. Asimismo, en el mantenimiento de infraestructuras, enjambres de robots pueden inspeccionar y reparar redes eléctricas o tuberías subterráneas, optimizando sus rutas para minimizar interrupciones en el servicio.

Las aplicaciones descritas demuestran la versatilidad y el potencial de los sistemas de planificación de movimiento en robótica multiagente, subrayando su relevancia en la mejora de procesos operativos, la eficiencia en la gestión de recursos y la respuesta efectiva en situaciones críticas. Con el continuo avance tecnológico, la adopción de estos sistemas seguirá expandiéndose, contribuyendo significativamente a la innovación y eficiencia en múltiples sectores.

REFERENCIAS

[1] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots, second edition*. Intelligent Robotics and Autonomous Agents series, MIT Press, 2011.

[2] N. Sindhwani, R. Anand, A. George, and D. Pandey, *Robotics and Automation in Industry 4.0: Smart Industries and Intelligent Technologies*. Big Data for Industry 4.0, CRC Press, 2024.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman Co., 1990.

[4] D. Zapata and G. Holguín, "Labyrinth project." https://github.com/DnelZpt/labyrinth, 2024. GitHub repository.

[5] Python, "tkinter — python interface to tcl/tk," 2024.

[6] Python, "threading — thread-based parallelism," 2024.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. THe MIT Press, 3rd. ed. ed., 2009.

[8] G. B. Dantzig, "The simplex method.," 1956.

[9] S. Mitchell, M. O'Sullivan, and I. Dunning, "Pulp : A linear programming toolkit for python," 2011.

[10] J. Forrest, T. Ralphs, S. Vigerske, H. G. Santos, J. Forrest, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, M. Lubin, Jan-Willem, rlougee, jpgoncal1, S. Brito, h-i gassmann, Cristina, M. Saltzman, tosttost, B. Pitrus, F. MATSUSHIMA, and to st, "coin-or/cbc: Release releases/2.10.11," 10 2023.

[11] J. Quintana, "Multi-agent robotics motion planning." https://github.com/JoseQuintana20/Motion_Planning, 2024. GitHub repository.