AGENTE INTELIGENTE PARA LABERINTO DINÁMICO

Michael Steven Rodriguez Arana

Yeifer Ronaldo Muñoz Valencia

Juan Carlos Rojas Quintero

Inteligencia Artificial

Joshua Rodriguez

Facultad de Ingeniería

Universidad del Valle Sede Tulua

AGENTE INTELIGENTE PARA LABERINTO DINÁMICO

1. Introducción

El presente informe documenta el desarrollo e implementación de un agente inteligente diseñado para navegar eficientemente en un laberinto dinámico, utilizando diversas técnicas de búsqueda y adaptándose a cambios en el entorno durante la ejecución.

El objetivo principal del proyecto es construir un sistema capaz de encontrar el camino óptimo desde un punto de inicio hasta un objetivo, donde las condiciones pueden variar en tiempo real: paredes y obstáculos pueden aparecer o desaparecer, y la ubicación del objetivo puede modificarse dinámicamente. Para ello, el agente implementa y combina técnicas de búsqueda como Búsqueda en Amplitud (BFS), Búsqueda en Profundidad (DFS), Búsqueda de Costo Uniforme (UCS), Búsqueda Avara (Greedy) y Búsqueda A*.

Una de las características fundamentales de este agente es su capacidad para adaptarse al entorno. Durante la ejecución, el agente evalúa continuamente su desempeño y el estado del laberinto, y en caso de detectar bloqueos o rutas ineficientes, puede cambiar de estrategia de búsqueda para mejorar su desempeño. Esta adaptación dinámica permite al agente reaccionar de manera inteligente ante escenarios impredecibles, garantizando una solución viable aun bajo condiciones adversas.

Este informe detalla las estructuras de datos utilizadas, la implementación de los algoritmos, y una batería de pruebas sobre distintos escenarios con variaciones en tamaño del laberinto, disposición de obstáculos, y comportamiento del objetivo.

- **Descripción del problema a resolver**

El problema a resolver en este proyecto consiste en diseñar un agente capaz de encontrar un camino válido y óptimo hacia un objetivo dentro de un laberinto cuyas condiciones no son estáticas.

A diferencia de escenarios tradicionales donde el entorno permanece constante, este laberinto presenta una serie de desafíos adicionales:

- **Cambios dinámicos en la estructura del entorno:** pueden aparecer o desaparecer paredes y obstáculos durante la ejecución del agente.

- **Ubicación variable del objetivo:** el queso puede cambiar de posición de manera aleatoria o programada a lo largo de la ejecución, lo que obliga al agente a recalcular rutas constantemente.

- **Dimensiones y configuraciones variadas del laberinto:** el sistema debe permitir definir entornos con diferentes tamaños y niveles de complejidad, lo cual impacta directamente en la estrategia de búsqueda.

- **Necesidad de adaptabilidad:** un único algoritmo de búsqueda no es suficiente para resolver todas las situaciones de forma eficiente. Por tanto, el agente debe ser capaz de cambiar de técnica según el estado del entorno, evaluando factores como bloqueo de rutas, tiempo de respuesta y costo acumulado.

El problema radica en lograr que el agente no sólo explore eficientemente el laberinto para alcanzar el objetivo, sino que lo haga de manera robusta y flexible, adaptándose a cambios imprevistos sin comprometer su funcionalidad ni eficiencia.

2. Descripción General del Sistema

2.1. Explicación general del sistema y ejecución

El flujo de ejecución general comienza con la inicialización del entorno y la configuración del laberinto, seguido por la activación del agente, quien ejecuta un algoritmo de búsqueda para encontrar el camino óptimo. Durante la ejecución, el entorno puede cambiar de forma aleatoria o programada, lo que obliga al agente a reevaluar su estrategia y, si es necesario, cambiar de algoritmo para adaptarse a las nuevas condiciones.

El proceso comienza en un menú interactivo, donde el usuario puede:

- Seleccionar el tamaño del laberinto (dimensiones personalizables).

- Elegir si desea que los obstáculos se ubiquen manualmente (haciendo clic sobre las celdas del entorno) o si se generan de forma aleatoria.

El flujo básico puede describirse así:

1. Se inicializa el entorno gráfico (usando Pygame).
2. Se carga el laberinto con obstáculos y elementos dinámicos.
3. El agente analiza el entorno y selecciona un algoritmo de búsqueda inicial.
4. El entorno puede cambiar (movimiento del queso, aparición de paredes).
5. El agente evalúa su progreso y, si es necesario, adapta su estrategia.
6. El proceso continúa hasta que el agente alcanza el objetivo o determina que no hay solución.

2.2. Tecnologías utilizadas

- **Python 3**: Lenguaje de programación principal del proyecto.
- **Pygame**: Usado para crear la interfaz gráfica del laberinto, visualizar al agente y representar los elementos dinámicos del entorno.
- **NetworkX**: Utilizado para modelar el laberinto como un grafo, facilitando la representación de nodos y aristas, así como la implementación de algoritmos sobre grafos.
- **Estructuras de datos personalizadas**: Para representar celdas, agentes, trampas y caminos con sus respectivos costos.

2.3. Estructura del código y componentes principales

El proyecto está organizado en diferentes módulos, lo que permite una mayor modularidad, reutilización de código y claridad en la lógica del sistema:

agent/

Contiene la lógica de los diferentes algoritmos de búsqueda:

- **agent.py** : Clase base del agente inteligente.
- **AdaptiveAgent.py**: Extiende al agente base e implementa la lógica de adaptación dinámica.
- **SearchAlgorithm.py**: Define una interfaz común para las técnicas de búsqueda.
- **bfs.py, dfs.py, ucs.py, greedy.py, aStar.py**: Implementaciones individuales de cada técnica de búsqueda.

laberinto/

Contiene la lógica del entorno:

- **cell.py:** Representación de cada celda del laberinto (pared, libre, trampa, etc.).
- **grid.py**: Modelo completo del laberinto.
- **Box.py:** Posible módulo visual o estructural relacionado con elementos interactivos.

gui/, configs/, fonts/, sources/

Módulos auxiliares para la interfaz gráfica, configuración y otros recursos.

main.py

Punto de entrada del programa. Se encarga de iniciar el entorno, cargar el laberinto, instanciar el agente y controlar el bucle principal de juego/interacción.

README.md, requirements.txt

Archivos para documentación del proyecto y gestión de dependencias.

3. Implementación del Agente Inteligente

3.1. Representación del entorno: Class Cell

La clase Cell es la unidad básica del laberinto, utilizada para representar cada celda dentro de la cuadrícula. Cada instancia de Cell mantiene información sobre su posición, estado, costo de movimiento, y características dinámicas como trampas o la presencia del queso.

Estructura de la clase Cell

- **Posición y estado**:
    - **Atributos x y y**: Coordenadas de la celda en la cuadrícula.
    - **visited**: Bandera para marcar si la celda ya ha sido explorada por el agente.
    - **is_traversed:** Marca si la celda ha sido parte del camino recorrido por el agente.
    - **traversed_color:** Color personalizado que permite una representación visual distinta de las celdas recorridas.
- **Paredes y movimientos**:
    - **walls:** Diccionario que indica si cada dirección (top, right, bottom, left) está bloqueada por una pared.
    - Métodos **is_wall()** y **set_wall()** permiten consultar y modificar estos bloqueos, facilitando la dinámica de apertura y cierre de pasadizos durante la ejecución.
- **Costos y trampas**:
    - **base_cost:** Costo base por atravesar la celda (por defecto 1).
    - **trap_type:** Puede ser "ratonera" o "gato", lo cual aumenta el costo de movimiento.
    - **trap_costs:** Diccionario con los valores adicionales que aportan estas trampas (+3 para ratonera y +5 para gato).
    - **cost:** Costo total de la celda, calculado como **base_cost + trap_cost** si hay trampa.

    - Métodos **set_trap() y remove_trap()** permiten añadir o quitar trampas dinámicamente durante el juego.

- **Objetivo y detección**:
    - **cheese:** Bandera que indica si la celda contiene el queso.
    - Métodos **is_cheese() y set_cheese()** manejan esta funcionalidad.

Importancia en la búsqueda

Esta clase es crucial para los algoritmos de búsqueda, ya que permite:

- Evaluar el costo dinámico de las rutas.
- Detectar trampas y evitarlas si es posible.
- Adaptar la estrategia del agente según la dificultad del terreno.

Además, su diseño permite una visualización clara en el entorno gráfico, ayudando a representar el estado del laberinto y el progreso del agente.

3.2. Lógica del Agente Inteligente: class Adaptative Agent

La clase **Adaptative Agent** implementa una solución de navegación dinámica dentro de un laberinto cambiante. Su objetivo es alcanzar una meta desde una posición inicial, ajustando su estrategia en tiempo de ejecución según el rendimiento de distintos algoritmos de búsqueda.

Inicialización del agente

El constructor de la clase **Adaptative Agent** recibe cuatro parámetros que definen el contexto inicial en el que operará el agente adaptativo dentro del entorno del laberinto. A continuación, se describe el propósito de cada argumento:

- **`Laberinto`**: representa la estructura del entorno donde se moverá el agente. Este objeto contiene la información del mapa del laberinto, como la distribución de celdas, la ubicación de trampas, y los costos asociados a cada tipo de terreno. Es utilizado por el agente para consultar el estado de las

celdas y planificar rutas dinámicas según las condiciones del entorno.

- **Start**: es una tupla que indica las coordenadas iniciales del agente dentro del laberinto, comúnmente en la forma (fila, columna). Define el punto de partida desde el cual el agente inicia su navegación.

- **Goal**: también es una tupla con formato (fila, columna), que señala la ubicación objetivo o destino final al que debe llegar el agente. Es utilizada como parámetro en la planificación del recorrido.

- **initial_algorithm** (opcional): es una cadena de texto que especifica el algoritmo de búsqueda con el que el agente debe iniciar su planificación. Los valores aceptados incluyen "A*", "UCS", "BFS", "Greedy", "DFS". Si no se proporciona este parámetro, el agente utiliza por defecto el algoritmo A* ("A*"). Este parámetro permite comparar el desempeño entre distintos algoritmos y establecer una estrategia inicial flexible.

```
class AdaptiveAgent:
    def __init__(self, laberinto, start, goal, initial_algorithm="A*"):
```

Atributos de la clase

self.**laberinto**: almacena la instancia del entorno (o mapa) sobre el cual el agente se desplaza. Este objeto permite consultar información de cada celda, como si contiene una trampa y el costo asociado al paso por dicha celda.

self.start y self.goal: contienen las coordenadas inicial y final, respectivamente, del recorrido del agente. Ambas están representadas como tuplas (fila, columna) y se utilizan como puntos de entrada y salida para los algoritmos de búsqueda.

self.current_algorithm: representa el nombre del algoritmo de búsqueda actualmente seleccionado. Puede tomar valores como "A*", "UCS", "BFS", "Greedy" o "DFS". Este atributo se actualiza si durante la ejecución el agente detecta que otro algoritmo tiene mejor rendimiento en el contexto actual.

self.algorithms: es una lista con los nombres de los algoritmos disponibles para la planificación. Esta lista es utilizada en la evaluación comparativa para determinar cuál estrategia ofrece mejores resultados ante cambios en el entorno.

self.agent: contiene una instancia de la clase Agent, que es responsable de ejecutar el algoritmo de búsqueda seleccionado. Esta instancia puede ser reemplazada al cambiar de algoritmo.

self.metrics: es un diccionario que almacena métricas clave sobre el comportamiento del agente. Incluye:

- replan_time: tiempo (en milisegundos) que toma recalcular una ruta.
- blocked_nodes: cantidad de nodos bloqueados (por trampas u obstáculos) encontrados durante la navegación.
- cost_history: lista con el historial de costos acumulados de cada ruta planificada.
- change_counter: número de veces que el agente ha cambiado de algoritmo durante la ejecución.

self.visited_cells: es una lista de todas las celdas que ha visitado el agente en su recorrido. Esta información puede ser útil para analizar el comportamiento del agente o para visualizaciones posteriores.

self.traversal_cost: almacena el costo total acumulado por el agente al desplazarse desde su punto de partida hasta su estado actual.

Selección del mejor algoritmo

```
def calculate_astar_score(self):
    time_score = max(0, 10 - self.metrics["replan_time"] * 0.1)
    step_score = max(0, 10 - self.agent.total_steps * 0.05)
    return time_score + step_score

def calculate_ucs_score(self):
    cost_score = max(0, 10 - self.agent.total_cost * 0.1)
    step_score = max(0, 10 - self.agent.total_steps * 0.05)
    return cost_score + step_score

def calculate_bfs_score(self):
    blocked_score = max(0, 10 - self.metrics["blocked_nodes"] * 0.5)
    step_score = max(0, 10 - self.agent.total_steps * 0.1)
    return blocked_score + step_score

def calculate_greedy_score(self):
    cost_score = max(0, 10 - self.agent.total_cost * 0.1)
    blocked_score = max(0, 10 - self.metrics["blocked_nodes"] * 0.5)
    return cost_score + blocked_score

def calculate_dfs_score(self):
    step_score = max(0, 10 - self.agent.total_steps * 0.05)
    cost_score = max(0, 10 - self.agent.total_cost * 0.05)
    return step_score + cost_score
```

El agente adaptativo utiliza un sistema de puntuación personalizado para evaluar el desempeño de cada algoritmo de búsqueda (A*, UCS, BFS, Greedy y DFS), con el objetivo de seleccionar dinámicamente el más adecuado según el entorno y la situación actual. Los puntajes se calculan en una escala de 0 a 20, combinando métricas clave asociadas a cada estrategia.

- A*

Este algoritmo se evalúa con base en el tiempo de recálculo y el número de pasos estimados hasta la meta:
time_score = max(0, 10 - replan_time * 0.1)
step_score = max(0, 10 - total_steps * 0.05)
Puntaje total = time_score + step_score

- UCS

El puntaje depende del costo acumulado de la ruta y de los pasos:
cost_score = max(0, 10 - total_cost * 0.1)
step_score = max(0, 10 - total_steps * 0.05)
Puntaje total = cost_score + step_score

- BFS

Evalúa la cantidad de nodos bloqueados y los pasos recorridos:
blocked_score = max(0, 10 - blocked_nodes * 0.5)
step_score = max(0, 10 - total_steps * 0.1)
Puntaje total = blocked_score + step_score

- Greedy

Este algoritmo se evalúa por el costo total y la presencia de obstáculos:
cost_score = max(0, 10 - total_cost * 0.1)
blocked_score = max(0, 10 - blocked_nodes * 0.5)
Puntaje total = cost_score + blocked_score

- DFS

Evalúa pasos y costo total, considerando que DFS suele explorar mucho:
step_score = max(0, 10 - total_steps * 0.05)
cost_score = max(0, 10 - total_cost * 0.05)
Puntaje total = step_score + cost_score

Selección del mejor algoritmo:
Una vez calculados los puntajes, el algoritmo con mayor puntuación es seleccionado. Si este puntaje supera levemente al del algoritmo actual (diferencia > 0.0001), se realiza el cambio de estrategia. Esta lógica permite que el agente se adapte al entorno en tiempo real, optimizando su desempeño según las condiciones del laberinto.

Selección del algoritmo y búsqueda

El método **find_path(start, goal)** es el encargado de ejecutar el algoritmo de búsqueda correspondiente. Dependiendo del valor actual de **self.algorithm**, se instancia la clase adecuada (**BFS, DFS, UCS, Greedy, AStar**), todas ellas definidas en el paquete **agent**.

La búsqueda produce:

- **self.path:** La secuencia de celdas desde el origen hasta el queso.
- **self.explored:** Celdas visitadas, incluso si no se encontró un camino.
- **self.total_cost:** Costo total del camino encontrado, si aplica.

```
def find_path(self, start, goal):
    """Encuentra el camino usando el algoritmo seleccionado"""
    if self.algorithm == "BFS":
        searcher = BFS(self.grid)
    elif self.algorithm == "DFS":
        searcher = DFS(self.grid)
    elif self.algorithm == "UCS":
        searcher = UCS(self.grid)       RonaldoMunoz, 4 days ago • Se implementaron los algoritmos de busque
    elif self.algorithm == "Greedy":
        searcher = Greedy(self.grid)
    else:  # A*
        searcher = AStar(self.grid)

    result = searcher.find_path(start, goal)
    self.path = result if result else []
    self.explored = getattr(searcher, "explored_cells", [])  # Guardar celdas exploradas si no hay camino

    self.current_step = 0
    self.total_steps = len(self.path) - 1 if self.path else 0
    self.total_cost = getattr(searcher, "final_cost", 0)
```

Movimiento del agente

Finalmente, el método get_next_move() permite al sistema recuperar el siguiente paso del camino para ser ejecutado en la interfaz gráfica. En caso de no haber un camino válido, recorre las celdas exploradas.

```python
def get_next_move(self):
    """Devuelve la siguiente posición en el camino o celdas exploradas si no hay camino"""
    if self.path and self.current_step < len(self.path):
        next_pos = self.path[self.current_step]
        self.current_step += 1
        return next_pos
    elif not self.path and self.current_step < len(self.explored):
        next_pos = self.explored[self.current_step]
        self.current_step += 1
        return next_pos
    return None
```

Esta clase demuestra un enfoque adaptativo e inteligente frente a entornos cambiantes, y es la responsable directa de que el agente se comporte de forma autónoma, reactiva y eficaz.

4. Logica del Archivo Principal (main.py)

El archivo main.py es el punto de entrada principal del juego. Orquesta la ejecución completa del laberinto, desde la configuración inicial hasta el despliegue visual del agente y su búsqueda inteligente del camino al queso. A continuación, se desglosan los componentes clave:

```python
from laberinto.Box import menu, setup_board, COLOR_FONDO, COLOR_PARED, MARGIN
from agent.AdaptiveAgent import AdaptiveAgent
import pygame
import sys
import random      Rojasjuan08, 13 hours ago • Implementacion de un entorno dinámico …
```

Estos módulos permiten: mostrar la interfaz de configuración (menu), generar el tablero (setup_board), renderizar colores, usar Pygame para visualización y gestionar la lógica del agente adaptativo.

6.2. Funciones auxiliares

- **mover_queso()**:
  Reubica el queso aleatoriamente a una celda libre del laberinto (sin trampas).

- **modificar_obstaculos() / eliminar_obstaculos()**:
  Introducen y eliminan obstáculos de manera dinámica durante la ejecución del juego, simulando un entorno no estático y cambiante.

6.3. Lógica del juego: juego()

Este es el núcleo del juego. A continuación, se explican sus fases principales:

a) Configuración inicial

- Se ejecuta menu() para obtener filas, columnas y modo de obstáculos.
- Luego, setup_board() genera el laberinto, carga las imágenes y posiciona al agente y al queso.
- Se inicializa el AdaptiveAgent, el cual escoge automáticamente el algoritmo más eficiente para planificar su ruta hacia el queso.

b) Planificación inicial

El agente selecciona y ejecuta una planificación con el mejor algoritmo:

c) Bucle principal

Se ejecuta continuamente mientras el juego esté activo:

- **Eventos de usuario**: Permite salir con `ESC` o cerrar la ventana.
- **Adaptación dinámica**:
    - Cada 5 iteraciones el queso se mueve a otra posición válida.
    - Cada 3 iteraciones se agregan obstáculos.
    - Cada 5 iteraciones se remueven obstáculos.
    - El agente replanifica su camino cada vez que cambia el entorno.
- **Movimiento del agente**:
  Cada 100 ms el agente avanza un paso según el camino planificado.
- **Detección de estancamiento**:
  Si el agente ya no puede avanzar, se muestra un mensaje de fallo.

d) Renderizado gráfico

Cada fotograma se actualiza visualmente:

- Se dibujan el agente, queso, trampas y celdas ya visitadas.

- Se muestran líneas de cuadrícula y estadísticas del juego:
    - Pasos dados
    - Costo total del recorrido
    - Algoritmo seleccionado dinámicamente por el agente

e) Victoria

Si el agente alcanza el queso, se muestra un mensaje de éxito y se reinicia el juego.

6.4. Función main()

Llama indefinidamente a juego() para permitir reinicios continuos del laberinto. También gestiona el cierre del programa.

5. Tecnicas de Busqueda Implementadas

Búsqueda por Amplitud (BFS)

La clase **BFS** extiende de una clase base común **SearchAlgorithm** y redefine el método **find_path(start, goal)**, el cual realiza un recorrido **en anchura** utilizando una cola FIFO (deque de Python).

```
    while queue:
        current_pos, path, total_cost = queue.popleft()

        # Marcar la celda como recorrida (visual)
        current_cell = self.grid.get_cell(current_pos)
        current_cell.make_traversed((135, 206, 250))  # Azul claro para BFS
Rojasjuan08, 10 hours ago • Implementacion para que se pinten las celdas d
        # Guardar la celda como explorada
        self.explored_cells.append(current_pos)

        if current_pos == goal:
            self.final_cost = total_cost
            return path
```

Cada elemento en la cola contiene:

- La posición actual **(current_pos).**
- El camino recorrido hasta esa celda **(path)**.

- El costo total acumulado hasta el momento **(total_cost).**

En cada iteración, el algoritmo:

- Extrae el primer elemento de la cola.
- Marca la celda como explorada visualmente (color azul claro).
- Guarda la celda en **self.explored_cells.**
- Verifica si ha llegado al objetivo.
- Encola todos los vecinos no visitados, sumando el costo de la celda correspondiente

```
for neighbor, move_cost in self.grid.get_weighted_neighbors(current_pos):
    if neighbor not in self.visited:
        self.visited.add(neighbor)
        queue.append((neighbor, path + [neighbor], total_cost + move_cost))
```

Búsqueda por Profundidad (DFS)

La clase **DFS** también hereda de la clase base **SearchAlgorithm** y redefine el método **find_path(start, goal)**, utilizando una pila (estructura LIFO) para priorizar la exploración de rutas más profundas primero:

```
while stack:
    current_pos, path, total_cost = stack.pop()

    current_cell = self.grid.get_cell(current_pos)
    current_cell.make_traversed((255, 0, 0))  # Rojo para DFS

    # Registrar la celda como explorada
    self.explored_cells.append(current_pos)

    if current_pos == goal:
        self.final_cost = total_cost      You, 10 hours ago • funcion
        return path
```

En cada iteración:

- Se extrae el último elemento de la pila.

- Se marca la celda como recorrida visualmente con color rojo.
- Se añade la celda a **self.explored_cells.**
- Se verifica si se ha alcanzado el objetivo.
- Se apilan los vecinos no visitados para exploración futura.

```
for neighbor, move_cost in self.grid.get_weighted_neighbors(current_pos):
    if neighbor not in self.visited:
        self.visited.add(neighbor)
        stack.append((neighbor, path + [neighbor], total_cost + move_cost))
```

Búsqueda de Costo Uniforme (UCS)

La clase **UCS** hereda de **SearchAlgorithm** e implementa el método **find_path(start, goal)**. Se basa en una cola de prioridad (implementada mediante **heapq**) para siempre expandir el nodo de menor costo acumulado:

```
def find_path(self, start, goal):
    open_list = []
    heapq.heappush(open_list, (0, start))  # Cola con el costo 0 para empezar
    g_cost = {start: 0}  # Costo acumulado para cada celda
    came_from = {}  # Diccionario para reconstruir el camino
    self.visited.clear()
    self.explored_cells.clear()
```

Durante la ejecución:

- Se extrae el nodo con menor costo acumulado de la cola.
- Se marcan y colorean las celdas exploradas de rosa.
- Se actualizan los costos acumulados en el diccionario **g_cost.**
- Se registra el camino mediante el diccionario **came_from** para su posterior reconstrucción.

```python
# Explorar los vecinos
for neighbor, move_cost in self.grid.get_weighted_neighbors(current):
    new_cost = g_cost[current] + move_cost  # Calcular el nuevo costo

    if neighbor not in g_cost or new_cost < g_cost[neighbor]:
        g_cost[neighbor] = new_cost
        came_from[neighbor] = current
        heapq.heappush(open_list, (new_cost, neighbor))  # Insertar vecino con su costo
```

Búsqueda Avara (Greedy)

La clase **Greedy** hereda de **SearchAlgorithm** y utiliza una **heurística de Manhattan** para estimar la distancia entre dos puntos:

```python
def heuristic(self, a, b):
    # Heuristica de Manhattan
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def find_path(self, start, goal):      RonaldoMunoz, 4 days ago • Se implementaron los algoritmos de busqueda
    open_list = []
    heapq.heappush(open_list, (self.heuristic(start, goal), 0, start, [start]))  # (heurística, costo, posición, camino)
    self.visited.clear()
    self.explored_cells.clear()
```

El algoritmo utiliza una **cola de prioridad** basada en el valor heurístico para seleccionar el siguiente nodo a explorar.

Durante la ejecución:

- Se expande el nodo con menor valor heurístico.
- Se marcan las celdas exploradas con color **violeta**.
- El camino se construye paso a paso, sin considerar los costos reales de las trampas o el terreno.

Búsqueda A*

En esta implementación, la clase **AStar** extiende **SearchAlgorithm** y utiliza:

- **g(n)**: El costo real desde el inicio hasta la celda actual.
- **h(n)**: Heurística de Manhattan

- **Criterio de exploración**: **f(n) = g(n) + h(n)**

- Encuentra el camino más corto considerando el costo del terreno y la distancia al objetivo.

- **Visualización**: Celdas recorridas se marcan en **dorado**. La visualización incluye animación **(pygame.time.delay(20)**) para mostrar el avance en tiempo real.

- **Reacción al entorno**: Si se detectan cambios (**self.grid.changed**), la búsqueda se cancela para replanificar.

6. Implementación de la Interfaz Grafica

El sistema cuenta con una **interfaz gráfica interactiva**, implementada completamente en **Pygame**, que permite al usuario configurar el entorno del laberinto antes de ejecutar el juego. Esta etapa de configuración está diseñada para ser flexible y accesible, permitiendo personalizar tanto las dimensiones del laberinto como la forma en la que se generan los obstáculos.

Entradas de texto

Se utilizan componentes personalizados (Box) para permitir la entrada de texto numérico en campos dedicados a definir el número de **filas** y **columnas** del laberinto. Estos inputs detectan la interacción del usuario (clics y teclado), validan que solo se ingresen números y actualizan dinámicamente el valor mostrado en pantalla.

Botones de selección de modo de obstáculos

Se presentan dos botones:

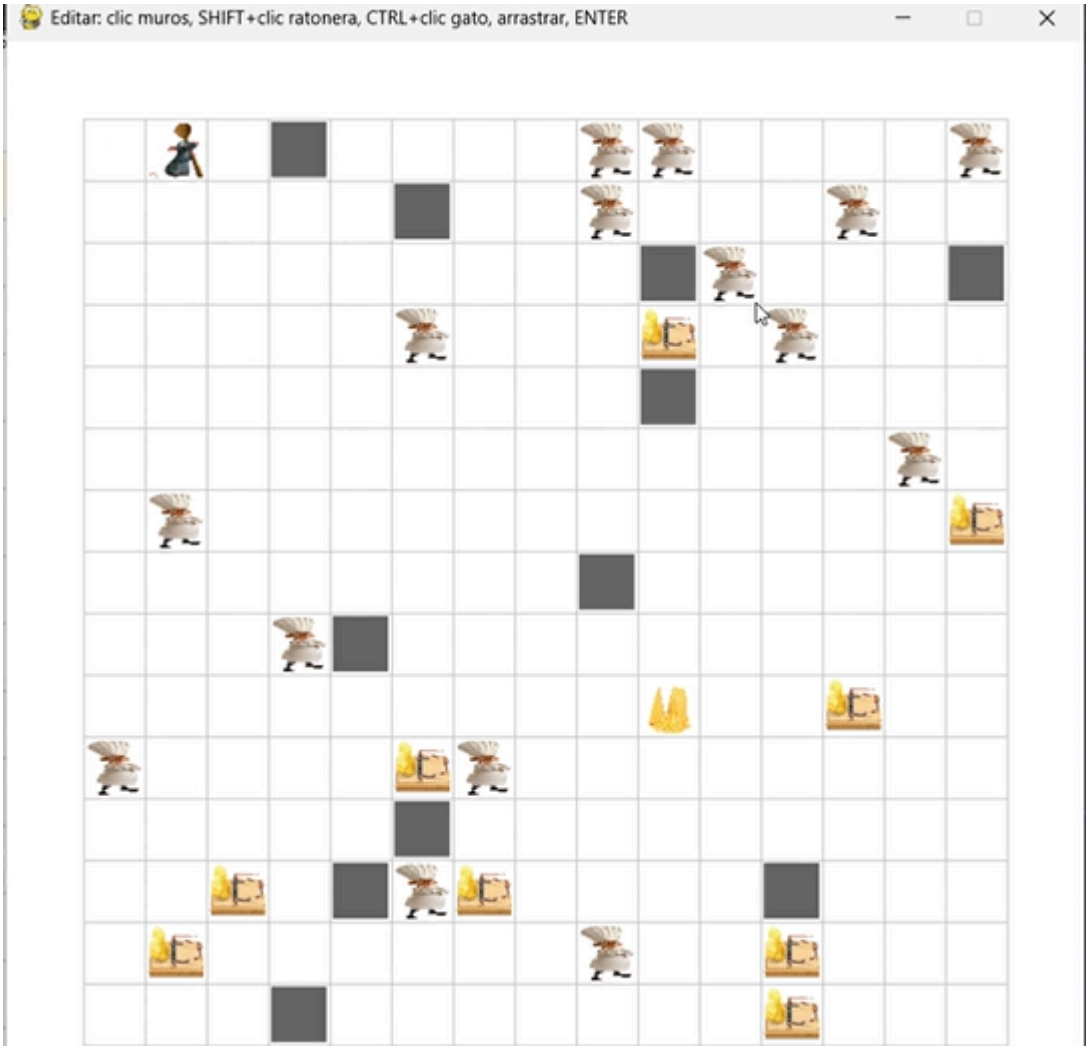
- **Manual**: el usuario puede colocar obstáculos de forma interactiva en el tablero, haciendo clic con combinaciones de teclas:
    - **Shift + clic**: coloca una **ratonera**.
    - **Ctrl + clic**: coloca un **gato**.
    - **Clic simple**: convierte una celda en pared (bloqueo total).
- **Aleatorio**: el sistema coloca automáticamente un número de obstáculos proporcional al tamaño del laberinto (aproximadamente 1/6 del total de celdas), distribuidos aleatoriamente, evitando las celdas de inicio y meta. Los tipos de obstáculos se asignan con cierta probabilidad (por ejemplo, 30% ratonera, 30% gato, 40% pared).

Experiencia sonora

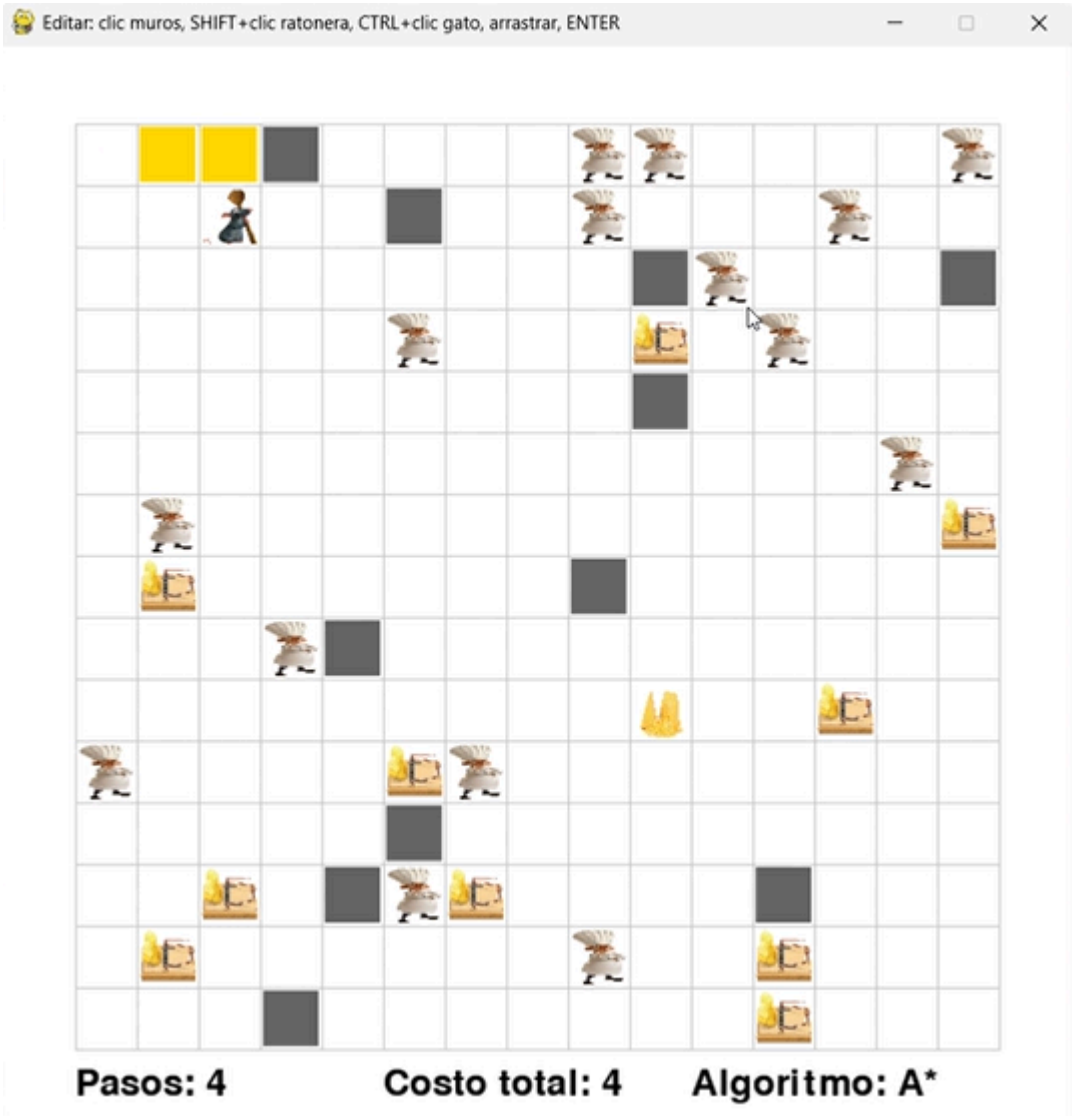
Como toque especial y para sumergir al usuario en la ambientación del juego, **al iniciar la interfaz de configuración suena una canción de la película *Ratatouille***. Esta se reproduce en bucle gracias al sistema de mezcla de audio de Pygame (**pygame.mixer.music**), lo cual enriquece la experiencia inmersiva del jugador desde el primer momento.

Pruebas del agente.

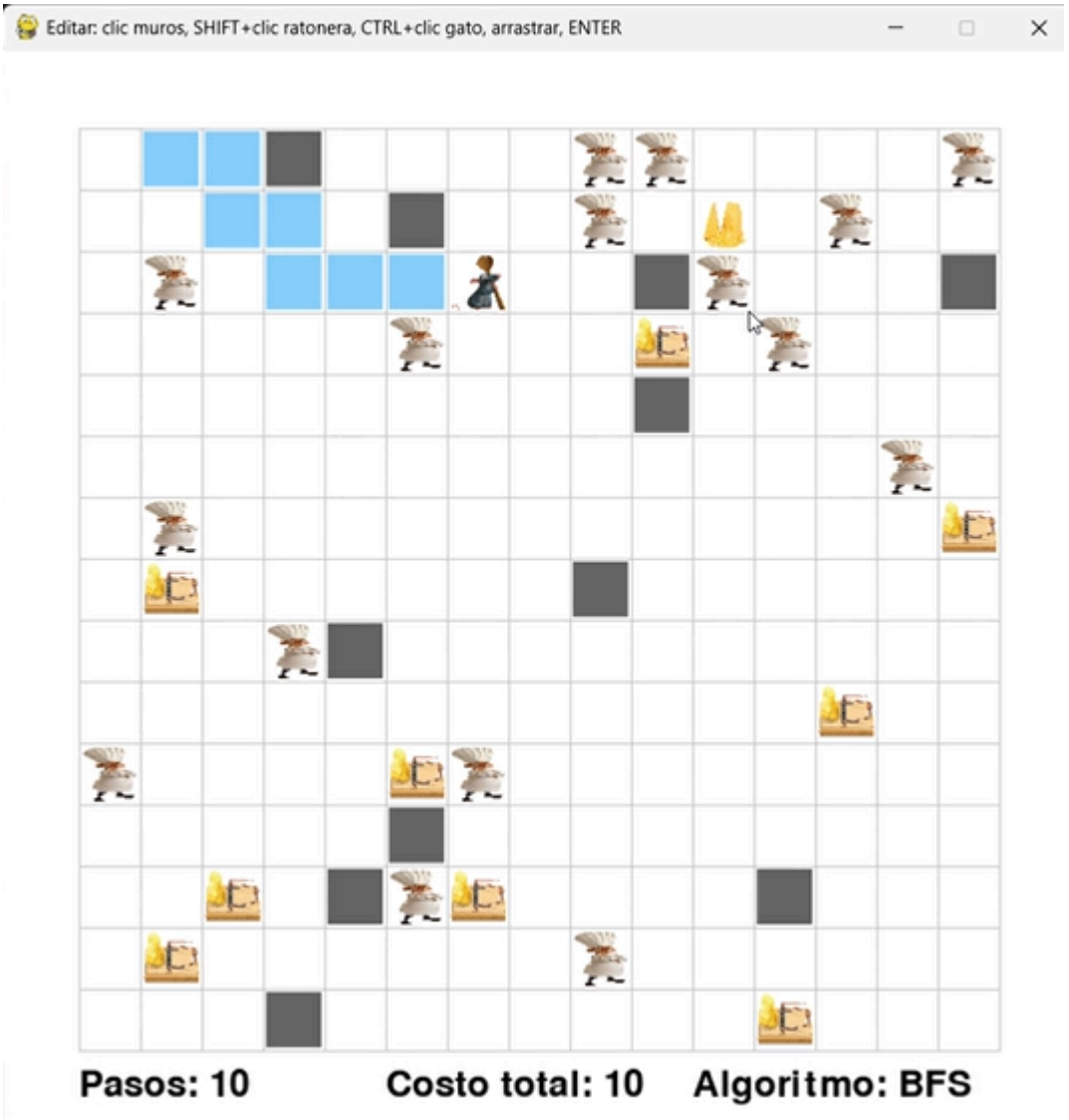
Estado inicial



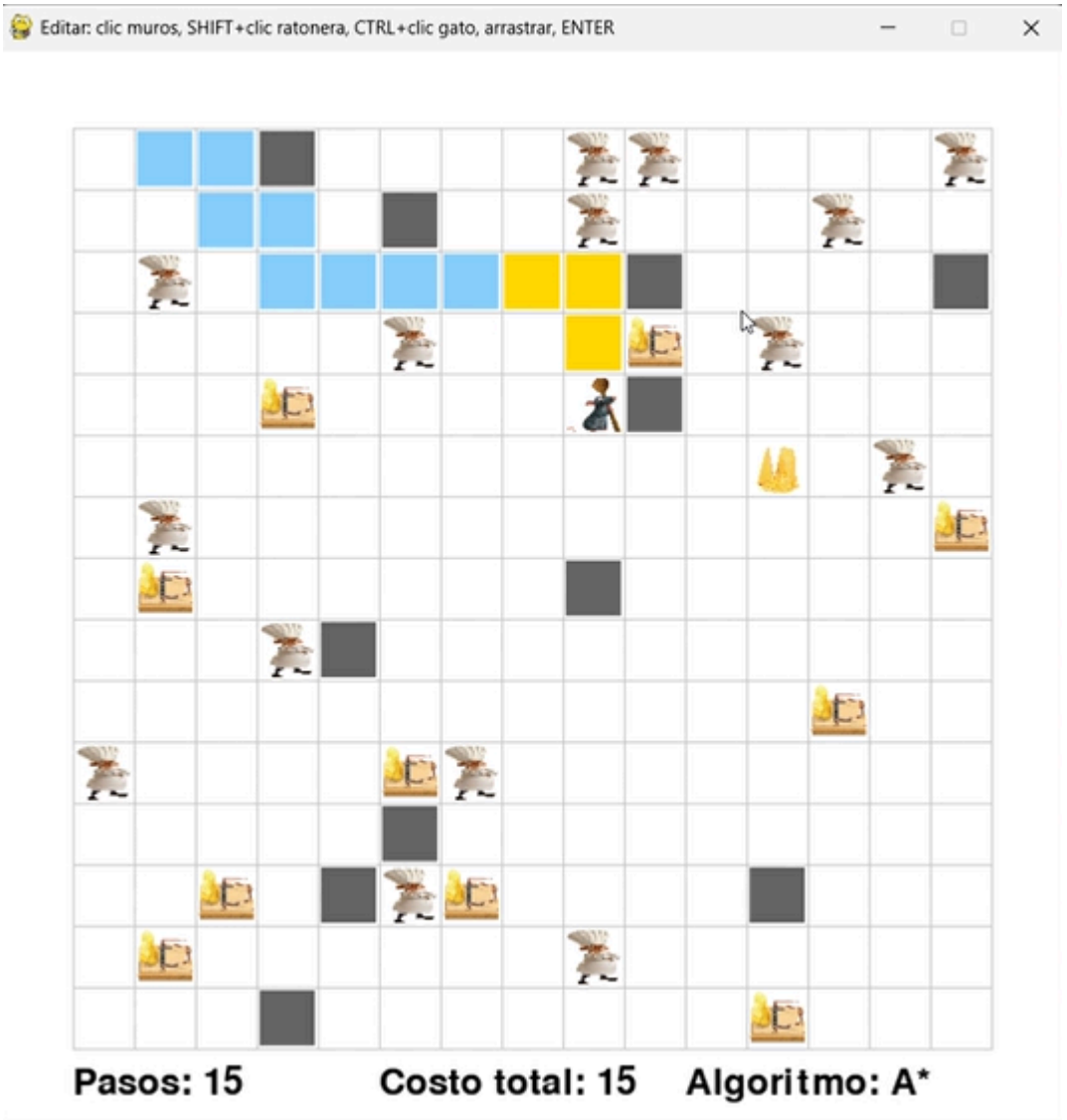
Primero utiliza el algoritmo A*



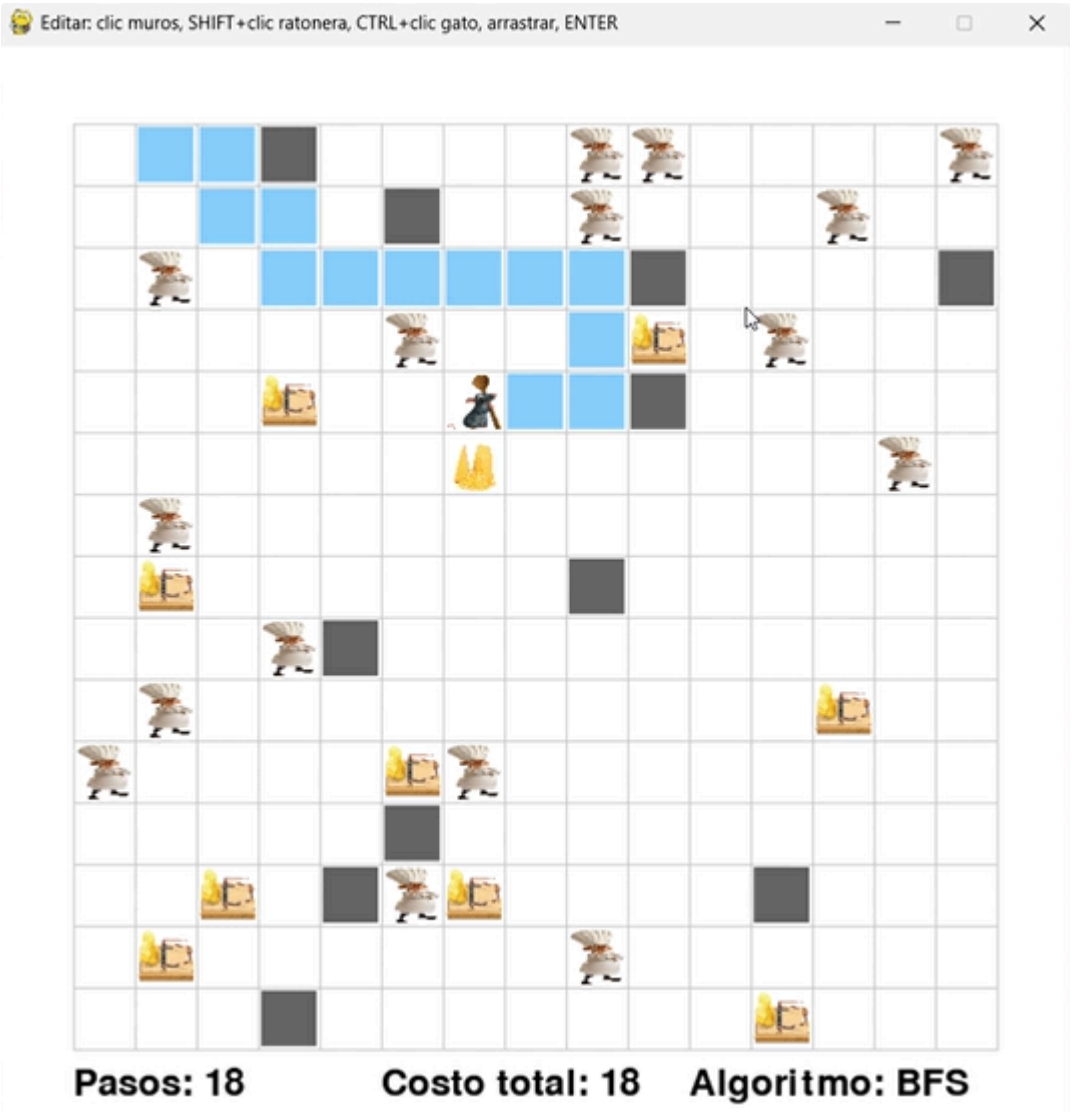
Cambia de algoritmo a BFS



al moverse el queso recalcula las rutas y vuelve a utilizar A*



el queso se mueve cerca del agente y este cambia a BFS



Por último el algoritmo encuentra el queso utilizando BFS y A*.


Editar: clic muros, SHIFT+clic ratonera, CTRL+clic gato, arrastrar, ENTER
¡Amigo el ratón del queso!
Pasos: 19
Costo total: 19
Algoritmo: BFS