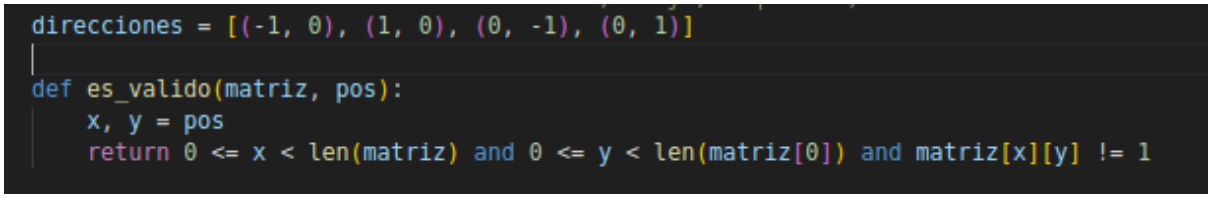
INFORME PROYECTO 1 IA

Alejandro Garzon Mayorga - 2266088
Andres Narvaez - 2259545
Kevin Alexis Lorza - 2266098

-Busqueda.py: implementación de los algoritmos de búsqueda

- búsqueda por amplitud (bfs)
- búsqueda por profundidad (dfs)
- costo uniforme (costo_uniforme)
- búsqueda avara (avara)
- A* (a_asterisco)

En el módulo de implementación de los algoritmos de búsqueda listados anteriormente se importan las clases deque y heapq para las estructuras de datos a utilizar en los algoritmos mencionados anteriormente



```
direcciones = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def es_valido(matriz, pos):
    x, y = pos
    return 0 <= x < len(matriz) and 0 <= y < len(matriz[0]) and matriz[x][y] != 1
```

En direcciones se declara los posibles movimientos que se puede tener estando ubicado en una casilla de la matriz

En la funcion es_valido se verifica que la posicion dada compuesta en fila, columna (x,y) es accesible dentro de la matriz, además el diferente a uno es para saber si es obstáculo o casilla vacía. (0 casilla vacía, 1 obstáculo)

-búsqueda por amplitud (bfs)

```python
#busqueda por amplitub
def bfs(matriz, inicio, objetivo):
    # Asegúrate de que el inicio y el objetivo sean válidos
    if not es_valido(matriz, inicio) or not es_valido(matriz, objetivo):
        return None, None  # Si no es válido, retornamos None para ambos valores

    visitado = set([inicio])
    cola = deque([(inicio, [inicio])])
    arbol = {}  # Para almacenar el árbol de búsqueda

    while cola:
        nodo, camino = cola.popleft()
        if nodo == objetivo:
            return camino, arbol
        for dx, dy in direcciones:
            vecino = (nodo[0] + dx, nodo[1] + dy)
            if es_valido(matriz, vecino) and vecino not in visitado:
                visitado.add(vecino)
                cola.append((vecino, camino + [vecino]))
                arbol[vecino] = nodo  # Guardamos de dónde vino el vecino

    return None, arbol  # Si no se encuentra camino, devolvemos None para ambos
```

Primeros se valida si el punto de inicio y el objetivo son puntos de partida válidos, es decir diferentes de 1, además que estos mismos puntos están dentro de los límites de la matriz.
Se tiene un conjunto para almacenar los conjuntos visitados. Se usa un conjunto ya que la búsqueda dentro de estos es muy eficiente en acceso ya que se implementan con las tablas hash y buscar dentro de estas tiene un costo de O(1).

Se utiliza una cola para almacenar el nodo inicio y una lista con el recorrido que lleva. Se usa una cola porque la inserción y eliminación en las colas son muy eficientes, por el algoritmo de organización de los nuevos elementos o los elementos a eliminar

Se utiliza un diccionario para almacenar el recorrido desde el inicio al objetivo. Esto porque los diccionarios son óptimos en el almacenamiento y búsqueda de elementos.

Mientras que la cola no esté vacía extrae la coordenada del nodo desde la cola. Se compara si es el objetivo para retornar el árbol y el camino. Sino empieza a recorrer cada uno de los caminos en las direcciones posibles en que se puede mover estando en un nodo, verificando si es válido para que esté en el rango de la matriz. Finalmente retorna el árbol del recorrido.

-búsqueda por profundidad (dfs)

```python
#busqueda por profundidad
def dfs(matriz, inicio, objetivo):
    # Asegúrate de que el inicio y el objetivo sean válidos
    if not es_valido(matriz, inicio) or not es_valido(matriz, objetivo):
        return None, None  # Si no es válido, retornamos None para ambos valores

    pila = [(inicio, [inicio])]
    visitado = set()
    arbol = {}

    while pila:
        nodo, camino = pila.pop()
        if nodo == objetivo:
            return camino, arbol
        if nodo in visitado:
            continue
        visitado.add(nodo)
        for dx, dy in direcciones:
            vecino = (nodo[0] + dx, nodo[1] + dy)
            if es_valido(matriz, vecino) and vecino not in visitado:
                pila.append((vecino, camino + [vecino]))
                arbol[vecino] = nodo  # Guardamos de dónde vino el vecino

    return None, arbol
```

Primero se verifica si el punto de inicio y el objetivo son válidos, es decir, que estén dentro de la matriz y no sean celdas bloqueadas. Luego se utiliza una pila para almacenar los nodos, ya que la búsqueda por profundidad explora primero los caminos más largos.

Se tiene un conjunto para marcar los nodos ya visitados y un diccionario para guardar el recorrido del árbol desde el nodo anterior.

En cada iteración, se saca el último nodo agregado a la pila, se verifica si es el objetivo, y si lo es, se retorna el camino junto con el árbol de recorrido. Si no lo es, se buscan los vecinos válidos en las direcciones posibles y se agregan a la pila.

En esta función se usa una pila para almacenar los nodos pendientes por recorrer, ya que por su principio LIFO, permite recorrer cada nodo hasta su máxima profundidad antes de pasar a los nodos de menos profundidad

-costo uniforme (costo_uniforme)

```python
def costo_uniforme(matriz, inicio, fin):
    filas = len(matriz)
    columnas = len(matriz[0])
    priority_queue = [(0, inicio, [inicio])] # (costo, nodo, camino)
    visitados = {inicio}
    arbol_busqueda = [inicio]

    while priority_queue:
        costo_actual, nodo_actual, camino_actual = heapq.heappop(priority_queue)
        if nodo_actual == fin:
            return camino_actual, arbol_busqueda

        fila, columna = nodo_actual
        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            nueva_fila, nueva_columna = fila + dr, columna + dc
            nuevo_nodo = (nueva_fila, nueva_columna)
            nuevo_costo = costo_actual + 1 # Costo de cada paso es 1 (puedes ajustarlo)

            if 0 <= nueva_fila < filas and 0 <= nueva_columna < columnas and matriz[nueva_fila][nueva_columna] != 1 and nuevo_nodo not in visitados:
                visitados.add(nuevo_nodo)
                nuevo_camino = list(camino_actual)
                nuevo_camino.append(nuevo_nodo)
                heapq.heappush(priority_queue, (nuevo_costo, nuevo_nodo, nuevo_camino))
                arbol_busqueda.append(nuevo_nodo)

    return None, arbol_busqueda
```

Esta búsqueda se expande el nodo con menor costo acumulado. Se utiliza una cola de prioridad que permite ordenar automáticamente los nodos según su costo. Se utiliza una cola de prioridad ya que permite agregar los nodos pendientes con costo acumulado, y dependiendo el costo acumulado, el que tenga menos será el que se visita dentro de la cola de prioridad.

Cada vez que se extrae un nodo, se valida si es meta (fin en este caso) para retornar el camino actual y el árbol de búsqueda. Si alguno no ha sido visitado, se calcula el nuevo costo para llegar a ese vecino y se vuelve a insertar en la cola.

Aquí se garantiza que siempre se visite primero el camino con menor costo total desde el inicio.

-búsqueda avara (avara)

```python
def avara(matriz, inicio, fin):
    filas = len(matriz)
    columnas = len(matriz[0])
    priority_queue = [(manhattan_distance(inicio, fin), inicio, [inicio])] # (heurística, nodo, camino)
    visitados = {inicio}
    arbol_busqueda = [inicio]

    while priority_queue:
        _, nodo_actual, camino_actual = heapq.heappop(priority_queue)
        if nodo_actual == fin:
            return camino_actual, arbol_busqueda

        fila, columna = nodo_actual
        for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            nueva_fila, nueva_columna = fila + dr, columna + dc
            nuevo_nodo = (nueva_fila, nueva_columna)

            if 0 <= nueva_fila < filas and 0 <= nueva_columna < columnas and matriz[nueva_fila][nueva_columna] != 1 and nuevo_nodo not in visitados:
                visitados.add(nuevo_nodo)
                nuevo_camino = list(camino_actual)
                nuevo_camino.append(nuevo_nodo)
                prioridad = manhattan_distance(nuevo_nodo, fin)
                heapq.heappush(priority_queue, (prioridad, nuevo_nodo, nuevo_camino))
                arbol_busqueda.append(nuevo_nodo)

    return None, arbol_busqueda
```

En esta el enfoque es llegar al objetivo usando una heurística (como la distancia de Manhattan), sin considerar el costo del camino recorrido.

Se usa una cola de prioridad ordenada por la heurística. A diferencia de la usada en costo uniforme esta permite que el algoritmo seleccione el nodo que aparentemente está más cercano, ya que no tiene en cuenta el costo acumulado, y el que actualmente tenga menor costo es el que será expandido

Finalmente se van agregando los vecinos válidos según qué tan cerca estén del objetivo, marcando los visitados y construyendo el árbol de búsqueda en el proceso.

-A* (a_asterisco)

```python
def a_asterisco(matriz, inicio, objetivo):
    # Asegúrate de que el inicio y el objetivo sean válidos
    if not es_valido(matriz, inicio) or not es_valido(matriz, objetivo):
        return None, None  # Si no es válido, retornamos None para ambos valores

    cola = [(0 + heuristica(inicio, objetivo), 0, inicio, [inicio])]
    visitado = set()
    arbol = {}

    while cola:
        f, costo, nodo, camino = heapq.heappop(cola)
        if nodo == objetivo:
            return camino, arbol
        if nodo in visitado:
            continue
        visitado.add(nodo)

        for dx, dy in direcciones:
            vecino = (nodo[0] + dx, nodo[1] + dy)
            if es_valido(matriz, vecino) and vecino not in visitado:
                nuevo_costo = costo + 1
                prioridad = nuevo_costo + heuristica(vecino, objetivo)
                heapq.heappush(cola, (prioridad, nuevo_costo, vecino, camino + [vecino]))
                arbol[vecino] = nodo  # Guardamos de dónde vino el vecino

    return None, arbol
```

Se define una cola la cual va a tener la heurística en cada nodo (elemento de la cola), el nodo donde inicia, el nodo actual y el camino al nodo en el que se está parado actualmente.

Posteriormente se valida si el nodo actual es el objetivo o no. Si no empieza a recorrer a sus vecinos en las direcciones posibles, pregunta si es válido el nodo vecino en la matriz y va aumentando el costo. Así mismo se va reconstruyendo el árbol con el camino desde el inicio.

Se usa una cola de prioridad, ya que cada nodo tendrá su costo basado en la heurística y el costo real, así mismo el que tenga el menos costo en la cola es el que será visitado.

-Modulo Interfaz.py
El Grafo (networkx):

- Complejidad: La creación y manipulación de grafos utilizando la librería networkx es un concepto avanzado en la representación de relaciones y la aplicación de algoritmos de grafos. En tu caso, el laberinto se representa como un grafo donde cada celda no-pared es un nodo y las celdas adyacentes tienen una arista entre ellas. El peso de las aristas se modifica si la celda vecina es un enemigo.
- Cuidado:
- Asegurarte de que la conversión de la matriz del laberinto al grafo sea correcta es fundamental. Un grafo mal construido no representará fielmente el laberinto y los algoritmos de búsqueda no funcionarán correctamente.
- Comprender cómo los pesos de las aristas afectan el comportamiento de los algoritmos de búsqueda basados en costos (como Costo Uniforme y A*) es importante. En tu caso, los enemigos aumentan el "costo" de pasar por esas celdas, lo que influirá en el camino elegido por el agente.
- Ejemplo (Función crear_grafo_desde_matriz):
  Python
  def crear_grafo_desde_matriz(current_matriz):
  G = nx.Graph()
  for i in range(FILAS):
  for j in range(COLUMNAS):
  if current_matriz[i][j] != 1: # Si no es una pared
  for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]: # Vecinos
  ni, nj = i + dx, j + dy
  if 0 <= ni < FILAS and 0 <= nj < COLUMNAS and current_matriz[ni][nj] != 1:
  peso = 1
  if current_matriz[ni][nj] == 2: # Si el vecino es un enemigo
  peso = 5
  G.add_edge((i, j), (ni, nj), weight=peso)
  return G

- Esta función itera a través de cada celda de la matriz. Si la celda no es una pared, considera sus cuatro vecinos (arriba, abajo, izquierda, derecha). Si un vecino está dentro de los límites del laberinto y no es una pared, se añade una arista entre la celda actual y el vecino al grafo G. El weight de la arista es 1 por defecto, pero se establece en 5 si el vecino es una celda que contiene un enemigo (valor 2 en la matriz).

3. Búsqueda del Camino (buscar_camino):

- Complejidad: Esta función actúa como un selector de la estrategia de búsqueda. Su complejidad radica en que delega la tarea real a las funciones de búsqueda individuales, cada una con su propia complejidad algorítmica.
- Cuidado:
- Asegurarte de que la función buscar_camino llame correctamente a la función de búsqueda seleccionada y que pase los argumentos adecuados (matriz o grafo, inicio, fin) es crucial.
- Entender que la eficiencia de la búsqueda del camino dependerá del algoritmo seleccionado y de las características del laberinto (tamaño, número de obstáculos, presencia de enemigos).
- Ejemplo (Función buscar_camino):
  Python
  def buscar_camino(metodo, current_matriz, inicio, fin):
  if metodo == "BFS":
  return bfs_matrix(current_matriz, inicio, fin)
  elif metodo == "DFS":
  return dfs(current_matriz, inicio, fin)
  elif metodo == "Costo Uniforme":
  return costo_uniforme(current_matriz, inicio, fin)
  elif metodo == "Avara":
  return avara(current_matriz, inicio, fin)
  elif metodo == "A*":
  return a_asterisco(current_matriz, inicio, fin)
  return None, None

- Explicación Detallada: Esta función toma el método de búsqueda como entrada (una cadena como "BFS" o "A*") y, basándose en este valor, llama a la función correspondiente del módulo Búsqueda. Pasa la current_matriz (o el grafo, dependiendo de cómo estén implementadas las funciones de búsqueda), la posición de inicio del agente y la posición fin del queso. Retorna el camino encontrado y, potencialmente, información adicional como el árbol de búsqueda (aunque actualmente no se utiliza).

4. Movimiento de los Enemigos (mover_enemigos):

- Complejidad: Esta función implementa un comportamiento de persecución para los enemigos, utilizando el mismo mecanismo de búsqueda de caminos que el agente. Esto implica ejecutar un algoritmo de búsqueda para cada enemigo en cada paso del

juego, lo que puede ser computacionalmente costoso, especialmente en laberintos grandes o con muchos enemigos.

- Cuidado:
- Considerar la eficiencia de esta función es importante. Realizar una búsqueda completa para cada enemigo en cada fotograma puede ralentizar significativamente el juego. Podrías explorar estrategias de movimiento más simples para los enemigos si el rendimiento se convierte en un problema.
- La lógica de cómo los enemigos eligen su próximo movimiento (tomando el primer paso del camino encontrado hacia el agente) es una decisión de diseño. Podrías implementar comportamientos más sofisticados si lo deseas.
- (Función mover_enemigos):

```
def mover_enemigos(pos_agente, current_matriz, current_metodo):
enemigos = [(i, j) for i in range(FILAS) for j in range(COLUMNAS) if
current_matriz[i][j] == 2]
nuevos_enemigos = []
for i, j in enemigos:
inicio = (i, j)
fin = pos_agente
camino, _ = buscar_camino(current_metodo, current_matriz, inicio, fin)
if camino and len(camino) > 1:
siguiente = camino[1]
if current_matriz[siguiente[0]][siguiente[1]] == 0:
current_matriz[i][j] = 0
nuevos_enemigos.append(siguiente)
else:
nuevos_enemigos.append((i, j)) # Si el siguiente está bloqueado, se queda quieto
else:
nuevos_enemigos.append((i, j)) # Si no se encuentra camino, se queda quieto
for x, y in nuevos_enemigos:
if current_matriz[x][y] == 0:
current_matriz[x][y] = 2
return current_matriz
```

- La función primero identifica las posiciones de todos los enemigos en la matriz. Luego, para cada enemigo, utiliza la función buscar_camino (con la misma técnica que el agente) para encontrar un camino hacia la posición actual del agente. Si se encuentra un camino de más de un paso, el enemigo intenta moverse al siguiente paso en ese camino (si la celda está vacía). Si no se encuentra un camino o el siguiente paso está bloqueado, el enemigo permanece en su posición actual. Finalmente, actualiza la matriz con las nuevas posiciones de los enemigos.

5. Lógica de Selección de la Técnica "Óptima" (Implementación Avanzada):

- Complejidad: La idea de que el agente aprenda cuál técnica es más eficiente basándose en el costo es un concepto básico de aprendizaje por refuerzo o adaptación. La implementación que sugerí involucra el mantenimiento de estadísticas (costo total y número de veces utilizado) para cada técnica y la selección de la técnica con el menor costo promedio.
- Cuidado:
- La estrategia de cuándo y cómo seleccionar la técnica "óptima" es crucial. Una selección prematura basada en pocos datos podría llevar a elegir una técnica subóptima para ciertos tipos de laberintos.
- Considerar cómo manejar la exploración inicial de las diferentes técnicas es importante. El agente necesita probar todas las técnicas lo suficiente antes de poder tomar una decisión informada sobre cuál es la mejor.
- La definición de "óptimo" en este caso se basa únicamente en el costo. Podrías considerar otros factores como el tiempo de búsqueda o la longitud del camino.
- Ejemplo (Lógica dentro de boton_reiniciar o después de llegar al queso):

-
  mejor_tecnica = None
  costo_promedio_minimo = float('int')
  for tecnica, total_costo in costos_por_tecnica.items():
  if contador_por_tecnica[tecnica] > 0:
  costo_promedio = total_costo / contador_por_tecnica[tecnica]
  if costo_promedio < costo_promedio_minimo:
  costo_promedio_minimo = costo_promedio
  mejor_tecnica = tecnica
  if mejor_tecnica:
  metodo_actual = mejor_tecnica
  print(f"Seleccionando la técnica más óptima: {metodo_actual}")
  else:
  print("No hay suficientes datos para seleccionar la técnica más óptima.")

  Se itera a través del diccionario `costos_por_tecnica`. Para cada técnica que se ha utilizado al menos una vez (verificando el `contador_por_tecnica`), calcula el costo promedio. Mantiene un registro de la técnica con el costo promedio más bajo hasta el momento y la selecciona como la `metodo_actual` para el siguiente intento. Si no hay suficientes datos (ninguna técnica se ha usado), se informa al usuario.

  Estos son los puntos de tu código que involucran conceptos más avanzados y requieren una comprensión cuidadosa. Asegurarte de que la lógica subyacente de los algoritmos de búsqueda, la representación del laberinto como un grafo y la estrategia de movimiento de los enemigos sean correctas es fundamental para el funcionamiento de la simulación.