Taller 1 inteligencia artificial

Integrantes

Karin Ximena Guerrero Herrera - 202317306

Manuela Victoria Ragua Perez - 202310825

Tatiana Luna Perez Suancha -201921109

Juan Fernando Vanegas – 202320303

Punto 1: DFS (Depth-First Search)

- **Complejidad de Tiempo:** En el peor caso teórico, la complejidad temporal de DFS es O(b^m), donde *b* representa el factor de ramificación y *m* la profundidad máxima de la búsqueda, ya que el algoritmo puede explorar ramas muy profundas antes de encontrar una solución. Sin embargo, en esta implementación se utiliza un conjunto de visitados, lo que garantiza que cada celda o estado se expanda como máximo una vez. Bajo esta condición, en espacios de estados finitos, la complejidad temporal se reduce a O(n), donde *n* es el número total de estados alcanzables. Esto evita ciclos y exploraciones redundantes, haciendo que el costo crezca linealmente con el tamaño del problema.
- **Complejidad de Espacio:** La complejidad espacial de DFS es O(b · m), donde *b* es el factor de ramificación y *m* la profundidad máxima alcanzada. Esto se debe a que el algoritmo solo necesita almacenar en memoria el camino actual dentro de la pila, junto con algunos nodos hermanos pendientes de exploración.
- **¿El algoritmo es completo?:** Es completo en grafos finitos cuando se incorpora detección de ciclos. En esta implementación, el uso de un conjunto de visitados garantiza que cada estado se expanda como máximo una vez. Esto evita que el algoritmo quede atrapado en bucles infinitos y asegura que, si existe una solución, eventualmente será encontrada.
- **¿Es óptimo?:** No es óptimo, ya que devuelve la primera solución que encuentra, sin garantizar que sea la de menor costo. En entornos con costos variables el algoritmo puede seleccionar un camino más caro simplemente porque lo encontró antes.

Punto 2: BFS (Breadth-First Search)

- **Complejidad de Tiempo:** La complejidad temporal de BFS es O(b^d), donde *b* es el factor de ramificación y *d* la profundidad de la solución. Esto se debe a que el algoritmo explora los nodos nivel por nivel, generando todos los estados de cada profundidad antes de avanzar. Como consecuencia, el número de nodos crece exponencialmente. Cuando *d* es grande, el costo computacional aumenta rápidamente
- **Complejidad de Espacio:** Es O(b^d), ya que el algoritmo debe almacenar en memoria todos los nodos de la frontera en la cola. BFS mantiene simultáneamente una gran cantidad de estados pendientes de expansión. Debido al crecimiento exponencial de la frontera, el consumo de memoria puede volverse muy alto.
- **¿El algoritmo es completo?:** BFS es completo en espacios de estados finitos con factor de ramificación finito. Gracias a su estrategia de exploración en anchura y al uso de una cola FIFO, el algoritmo garantiza que todos los nodos de profundidad *d* se expandan antes de avanzar al nivel *d + 1*. Esto asegura que, si existe una solución, será encontrada.
- **¿Es óptimo?:** Es óptimo. Garantiza encontrar el camino de menor número de pasos cuando todos los movimientos tienen el mismo costo. Sin embargo, no es óptimo en costo total si existen costos variables, ya que prioriza la profundidad mínima y no el costo acumulado.

Punto 3: Búsqueda de Costo Uniforme

- **Complejidad en Tiempo:** Es exponencial, es decir O(b^(1 + c/e)) el tiempo crece rápido porque el algoritmo explora un montón de rutas al mismo tiempo. Aquí b son las opciones de movimiento del robot, c es el costo total del camino ganador y e es la batería mínima que gasta dar un paso. Como revisa con mucho detalle cuánta batería gasta cada opción, le cuesta el procesamiento descartar las rutas malas.
- **Complejidad en Espacio:** También es exponencial, O(b^(1 + c/e)) porque el programa gasta demasiada memoria, debido a que la máquina tiene que guardar al mismo tiempo todas las rutas que va probando para poder ordenarlas de la más barata a la más cara y eso la llena de una.
- **¿El algoritmo es completo?** Sí, porque dar cualquier paso siempre gasta batería, el costo de las rutas malas siempre va a ir subiendo, debido a esto nunca se va a quedar atrapado en un ciclo infinito y siempre va a seguir buscando por el mapa hasta encontrar al sobreviviente.

- **¿Es óptimo?** Sí porque el sistema evalúa y ordena todo buscando siempre el menor consumo de energía, el primer camino que logre llegar al sobreviviente va a ser obligatoriamente el más económico.

Punto 4:

- **Complejidad temporal :** $O(b^d)$
- **Complejidad espacial:** $O(b^d)$
- **El algoritmo es completo,** ya que recorre todos los vecinos desde el estado inicial, e ignorando si es óptimo o no, si se puede llegar del estado inicial al final por algún camino, lo encontrará.
- **El algoritmo además también es óptimo,** debido a que la heurística (ya sea la distancia de Manhattan o la distancia Euclidiana) es admisible, entonces bajo estas condiciones si es óptimo el algoritmo, ya que siempre la h(n) va a ser menor o igual al costo real h*(n).
- **Constancia**: en el mejor de los casos, el moverse de una casilla a otra tiene el costo de 1, y los algoritmos consistentes tienen la característica que $h(n) \leq c(n,n')+h(n')$. El costo h(n) si estamos en (1,2) y el ciudadano está en (1,3) en Manhattan es de 1 y el Euclidiano también. En el mejor de los casos el c(n,n') sería de 1 también y el h(n') de 0, ósea el resultado es menor o igual a c(n,n')+h(n'), siendo entonces el algoritmo consistente.

Punto 5:

- **Complejidad temporal :** $O(b^d)$
- **Complejidad espacial:** $O(bd)$
- Se implementó A* sobre el problema de múltiples sobrevivientes usando como heurística la distancia Manhattan al sobreviviente más cercano. En el peor caso, el tiempo es O(b^d), donde b es el factor de ramificación y d la profundidad de la solución. El espacio de búsqueda crece considerablemente porque el estado incluye tanto la posición del agente como la configuración de sobrevivientes restantes, lo que hace que el número de estados sea exponencial respecto a la cantidad de sobrevivientes.
- En memoria, A* también tiene complejidad O(b^d), ya que mantiene los nodos generados en una cola de prioridad junto con el conjunto de visitados.
- El algoritmo es completo siempre que los costos de transición sean positivos y el espacio de búsqueda sea finito. Además, es óptimo porque la heurística

utilizada (distancia Manhattan al sobreviviente más cercano) es admisible, es decir, nunca sobreestima el costo real hasta la meta.

- La heurística también es consistente, ya que en una grilla se cumple que h(n) ≤ c(n,n') + h(n'). Al moverse una casilla, la distancia Manhattan cambia como máximo en 1, y el costo de movimiento es 1, por lo que se mantiene la desigualdad. Esto garantiza que A* conserve la optimalidad sin necesidad de reabrir nodos.
- En instancias pequeñas el algoritmo responde rápido, pero a medida que aumenta el número de sobrevivientes, el número de nodos expandidos crece significativamente. Esto sucede porque la heurística solo considera el sobreviviente más cercano y no incorpora información sobre las distancias entre los demás, lo que limita su capacidad de guiar eficientemente la búsqueda en problemas más grandes.

Reflexión punto 6b

A lo largo del taller, la Inteligencia Artificial fue una guía clave para conectar la teoría de algoritmos y heurísticas a la práctica. Mi versión inicial de la survivorHeuristic, por ejemplo, tenía fallas conceptuales y de sintaxis: no aplicaba correctamente la función min() y la fórmula de distancia en la grilla estaba mal planteada.

La interacción con la IA consistió en pedirle que prompts escalonados. Empezamos por lo general ("¿cómo construir una heurística para un grid?"), pasé a lo aplicado ("¿cómo se calcula la distancia Manhattan?") y finalmente pedí una revisión de mi fórmula. Las correcciones fueron puntuales pero invaluables; aprendí desde la lógica matemática hasta detalles técnicos como inicializar min_dist = float('inf'). Este proceso no sustituyó mi aprendizaje, sino que lo potenció: me permitió concentrarme en entender el comportamiento del algoritmo A* en lugar de frustrarme por errores de sintaxis que bloqueaban el simulador.