UNIVERSIDAD SANTO TOMÁS
PRIMER CLAUSTRO UNIVERSITARIO DE COLOMBIA
FACULTAD INGENIERÍA ELECTRÓNICA

Lab No. 1 - Introducción IA

Yefri Stiven Barrero Solano - 2320392
Universidad Santo Tomás – Sede Principal
Ingeniería Electrónica
Bogotá D.C

I. 8_Puzzle.py

El código presenta una implementación completa del algoritmo A* para resolver el clásico problema del 8-puzzle, un rompecabezas deslizante donde el objetivo es reorganizar ocho fichas numeradas en una cuadrícula de 3x3 utilizando el espacio vacío disponible. La solución combina elementos fundamentales de la inteligencia artificial y la teoría de búsqueda, ofreciendo una demostración práctica de cómo los algoritmos informados pueden resolver problemas de optimización complejos de manera eficiente. En este código se encontrará una estructura organizada que comienza verificando la posibilidad de solución mediante el análisis de paridad de inversiones, un concepto matemático crucial que determina si existe una secuencia válida de movimientos entre dos configuraciones. Esta verificación inicial evita intentos infructuosos en puzzles irresolubles, lo que representa una optimización importante.

Para esta solución se utiliza la heurística de distancia Manhattan, una función admisible que nunca sobrestima el costo real hacia la solución, garantizando así la optimalidad del camino encontrado. La implementación incluye una gestión eficiente de estados mediante una cola de prioridad que expande sistemáticamente los nodos más prometedores, balanceando inteligentemente entre el costo acumulado y la estimación futura. El código genera todos los movimientos válidos desde cualquier estado, considerando las restricciones físicas del tablero, y reconstruye la secuencia solución una vez alcanzado el objetivo. La representación visual de los estados mediante una función dedicada facilita el seguimiento del proceso, mostrando las configuraciones en un formato reconocible donde el espacio vacío se representa adecuadamente.

Un aspecto notable es el uso de estructuras de datos inmutables (tuplas) para representar estados, permitiendo su uso como claves en diccionarios y conjuntos, lo que optimiza las operaciones de búsqueda y comparación. El mecanismo de desempate mediante un contador incremental asegura un funcionamiento estable del heap en casos donde múltiples estados comparten el mismo valor de prioridad. El código incluido proporciona un caso de uso concreto con estados iniciales y objetivos específicos, permitiendo verificar inmediatamente el funcionamiento del algoritmo. La salida detallada muestra no solo la solución final sino cada paso intermedio, lo que resulta invaluable para fines educativos al revelar el proceso de toma de decisiones del algoritmo.

Esta implementación servirá como una excelente herramienta pedagógica para comprender los algoritmos de búsqueda informada, los conceptos de heurísticas admisibles y la resolución sistemática de problemas de espacios de estado complejos. El código está estructurado de manera que pueda extenderse fácilmente a variantes del problema o adaptarse para incorporar heurísticas alternativas, convirtiéndolo en una base versátil para una mayor exploración en algoritmos de búsqueda.

II. Problema_Lampara.py

El código implementa un sistema de control básico para una lámpara que puede encontrarse en dos estados posibles: "ENCENDIDA" o "APAGADA". El objetivo del sistema es transitionar desde un estado inicial de apagado hacia un estado meta de encendido mediante la ejecución de acciones específicas.

La lámpara opera bajo un modelo de estados finitos donde las acciones "PRENDER" y "APAGAR" están siempre disponibles regardless del estado actual, lo que significa que el usuario puede intentar prender o apagar la lámpara en cualquier momento. La función `cambiar_estado` gestiona las transiciones: cuando se ejecuta la acción "PRENDER", la lámpara pasa al estado "ENCENDIDA", mientras que la acción "APAGAR" la lleva al estado "APAGADA". Este comportamiento es determinista y no considera condiciones adicionales como fallos en la operación o estados intermedios.

La simulación comienza con la lámpara en estado "APAGADA" y ejecuta la acción "PRENDER", lo que resulta en una transición exitosa al estado "ENCENDIDA". El sistema verifica entonces si se ha alcanzado el estado meta, confirmando que el objetivo se ha cumplido satisfactoriamente. Este ejemplo simple ilustra los conceptos fundamentales de máquinas de estados y acciones discretas, proporcionando una base que podría extenderse para incorporar más complejidad, como comportamientos condicionales o estados adicionales.

III. Mascota_Virtual.py

El código implementa un sistema básico de estados y acciones para una mascota virtual, representando un ejemplo sencillo de máquina de estados finitos donde las transiciones entre estados están determinadas por acciones específicas. La mascota puede encontrarse en dos estados emocionales: "CONTENTO" o "TRISTE", con un estado inicial de tristeza y un estado meta de felicidad.

El desarrollo del código reside en la función `acciones_disponibles`, que define qué acciones son posibles según el estado actual de la mascota. Cuando está triste, la única acción disponible es "DAR_COMIDA", mientras que cuando está contenta, la acción permitida es "NO_HACER_NADA", reflejando un comportamiento donde la mascota solo requiere intervención cuando está infeliz. La función `cambiar_estado` gestiona las transiciones entre estados: alimentar a una mascota triste la convierte en contenta, mientras que no hacer nada con una mascota contenta mantiene su estado actual. Este diseño simple pero efectivo modela la relación causa-efecto entre acciones y estados emocionales.

La simulación demuestra el flujo completo del sistema, comenzando desde el estado inicial, mostrando las acciones disponibles, ejecutando una acción seleccionada y verificando si se alcanzó el estado meta. En este caso específico, la acción de dar comida resulta suficiente para transformar a la mascota de triste a contenta, alcanzando así el objetivo deseado.

Este código sirve como una introducción accesible a los conceptos de modelos de estados finitos y sistemas de transición de estados, mostrando cómo las acciones pueden afectar el estado de un ente virtual. Aunque minimalista en su implementación, proporciona una base que podría extenderse fácilmente con más estados emocionales (como "HAMBRIENTO" o "ENFERMO"), acciones adicionales (como "JUGAR" o "DAR_MEDICINA"), o incluso incorporar un sistema de necesidades más complejo con múltiples variables que afecten el estado emocional de la mascota.

IV. Buscando_Tesoro.py

En este caso se implementa una simulación simple de búsqueda de tesoro en un mundo grid 3x3, donde un pirata debe navegar desde una posición inicial hasta la ubicación del tesoro mediante una secuencia predefinida de movimientos. El entorno está delimitado por coordenadas que van desde (0,0) hasta (2,2), representando una cuadrícula compacta donde cada movimiento traslada al pirata una unidad en una

dirección cardinal. La simulación comienza definiendo el estado inicial del pirata en la esquina inferior izquierda (0,0) y la posición del tesoro en la esquina superior derecha (2,2). Las acciones posibles incluyen movimientos hacia arriba, abajo, izquierda y derecha, cada una asociada a un cambio específico en las coordenadas. La función `dentro_grid` verifica que cualquier movimiento propuesto mantenga al pirata dentro de los límites del mundo, evitando así desplazamientos inválidos.

Para cada estado, la función `acciones_disponibles` calcula dinámicamente los movimientos permitidos basándose en la posición actual, lo que permite adaptarse a los bordes del grid donde algunas direcciones estarían bloqueadas. La función `cambiar_estado` aplica una acción seleccionada, actualizando la posición del pirata solo si el movimiento es válido; de lo contrario, mantiene el estado actual sin cambios.

De esta manera, se ejecuta una secuencia predefinida de acciones (en este caso, ["ARRIBA", "ARRIBA", "DERECHA", "DERECHA"]), mostrando paso a paso la posición resultante después de cada movimiento. Si durante esta secuencia el pirata alcanza la posición (2,2), se declara que el tesoro ha sido encontrado y finaliza la ejecución; de lo contrario, al terminar la secuencia, se indica que el tesoro no fue alcanzado. Este enfoque permite probar fácilmente diferentes rutas modificando la lista de acciones, ofreciendo una herramienta básica pero efectiva para experimentar con planificación de caminos en entornos discretos y restringidos.

V. Laberinto_2x2.py

Para dar solución se implementa un sistema de búsqueda de rutas en un laberinto de dimensiones 2x2, donde un agente debe navegar desde un punto de inicio hasta un objetivo utilizando únicamente movimientos hacia la derecha y hacia abajo. La solución combina una representación formal de espacios de estados con algoritmos de búsqueda clásicos, ofreciendo una base sólida para entender conceptos fundamentales de inteligencia artificial y planificación. El espacio de estados está definido por todas las coordenadas posibles dentro de la grid [(0,0), (1,0), (0,1), (1,1)], con un estado inicial en (0,0) y un estado meta en (1,1). Las acciones permitidas son limitadas intencionalmente a movimientos hacia la derecha (incremento en x) y hacia abajo (incremento en y), creando un entorno restringido que simula restricciones de movimiento comunes en problemas de laberintos.

Es importante destacar el uso de la tabla de transiciones, que se construye dinámicamente considerando posibles obstáculos. Para cada par (estado, acción), la tabla especifica si el movimiento resulta en un nuevo estado válido, se encuentra bloqueado por un obstáculo, o lleva al agente fuera de los límites del grid. Esta implementación permite una gestión flexible de diferentes configuraciones de laberinto mediante la simple adición o remoción de obstáculos en el conjunto correspondiente.

El algoritmo de búsqueda empleado es Breadth-First Search (BFS), que garantiza encontrar el camino más corto (en número de movimientos) desde el inicio hasta el objetivo cuando existe una ruta válida. El BFS explora metódicamente todos los estados alcanzables, reconstruyendo la secuencia óptima de movimientos mediante un diccionario de padres que registra el estado predecesor y la acción utilizada para llegar a cada nuevo estado. El código incluye un sistema de recompensas básico que asigna un costo por paso (-1) y una recompensa por alcanzar la meta (+10), aunque este sistema no es utilizado activamente por el algoritmo de búsqueda en la implementación actual. Además, se proporcionan funciones de visualización que muestran claramente la tabla de transiciones y las rutas encontradas, facilitando el entendimiento del proceso de búsqueda.

La demostración incluye dos escenarios contrastantes: uno sin obstáculos donde se encuentra una ruta directa, y otro con un obstáculo en (0,1) que fuerza una reevaluación de las posibles rutas. Esta estructura permite observar cómo el algoritmo se adapta a diferentes configuraciones del entorno y encuentra soluciones óptimas incluso en presencia de restricciones. Esta implementación sirve como

una excelente introducción a los problemas de planificación de rutas, mostrando de manera concisa cómo los algoritmos de búsqueda sistemática pueden resolver problemas de espacios de estado discretos. La modularidad del código permite fácil extensión a grids más grandes, acciones adicionales o diferentes algoritmos de búsqueda, haciendo una herramienta con gran valor educativo para entender conceptos fundamentales de IA.

VI. Laberinto_3x3.py

Finalmente en esta solución se implementa un sistema de búsqueda de rutas en un laberinto de dimensiones 3x3, expandiendo significativamente las posibilidades de exploración en comparación con la versión 2x2. La solución mantiene una estructura similar pero introduce importantes mejoras en flexibilidad y visualización, representando un avance natural en complejidad para problemas de planificación de rutas. El espacio de estados ahora abarca nueve posiciones diferentes [(0,0) a (2,2)], con un punto de inicio en (0,0) y una meta en (2,2). Las acciones permitidas se han expandido para incluir los cuatro movimientos cardinales (arriba, abajo, izquierda, derecha), lo que permite una navegación más compleja y abre la posibilidad de crear caminos alternativos y soluciones más elaboradas.

El sistema construye una tabla de transiciones completa que considera tanto los límites del grid como obstáculos personalizables. Para cada combinación de estado y acción, la tabla determina si el movimiento es válido, está bloqueado por un obstáculo o llevaría al agente fuera de los límites permitidos. Esta implementación proporciona una base sólida para modelar diferentes configuraciones de laberintos con variados patrones de obstáculos. El algoritmo de búsqueda empleado sigue siendo Breadth-First Search (BFS), que garantiza encontrar el camino más corto en número de movimientos. El BFS explora sistemáticamente todos los estados alcanzables, reconstruyendo la ruta óptima mediante un registro de estados predecesores y las acciones que llevaron a cada nuevo estado.

Una mejora significativa en esta versión es la función de visualización del mapa (`imprimir_mapa`), que muestra gráficamente la configuración del laberinto utilizando símbolos intuitivos: 'S' para el inicio, 'G' para la meta, '#' para obstáculos y '.' para celdas libres. Esta representación visual facilita enormemente la comprensión de la estructura del problema y los desafíos de navegación.

El código incluye tres escenarios de prueba progresivos:

1. Un laberinto sin obstáculos donde se demuestra la ruta más directa
2. Un laberinto con un obstáculo en (0,1) que fuerza un desvío
3. Un escenario más complejo con obstáculos en (0,1) y (1,0) que bloquea las rutas cortas y requiere una solución más elaborada

Aunque se define un sistema de recompensas (coste por paso y recompensa por alcanzar la meta), este no se utiliza activamente en el algoritmo de búsqueda actual, sugiriendo que el código está preparado para extensiones futuras que implementen algoritmos de aprendizaje por refuerzo.

Esta implementación representa una excelente herramienta educativa para entender problemas de planificación de rutas en espacios de estado más grandes, mostrando cómo los algoritmos de búsqueda sistemática pueden adaptarse a entornos más complejos. La modularidad del diseño permite fácil extensión a grids de mayores dimensiones, diferentes algoritmos de búsqueda o la incorporación de elementos adicionales como costos variables por movimiento o tipos de terreno diferenciados.