Modelado por espacios de estados:

8-Puzzle, Lámpara, Mascota, Tesoro y Laberinto

Paula Sandoval

05 de Septiembre de 2025

Resumen

Este documento amplía la explicación conceptual y práctica de varios ejercicios de problemas de búsqueda haciendo uso de *espacio de estados + acciones + meta*. Para cada ejemplo se muestra: detalles del código (estructuras de datos y funciones clave) y teória implementada.

Índice

1. Resumen

Se presentan las siguientes propiedades que se utilizan en el desarollo de los ejercicios propuestos:

Estado: representación de la situación del sistema (por ejemplo, posición del sujeto o configuración del tablero).

Acción: transición de estado (ej. “mover arriba”, “prender”).

Función de transición: determinista o estocástica.

Costo: gasto asociado a ejecutar una acción

Heurística $h(s)$: estimación del costo restante desde los diferentes estados hasta la meta.

A*: busca minimizar $f(s) = g(s) + h(s)$, donde $g(s)$ es el costo desde el inicio hasta s. Si h es admisible y consistente, A* encuentra una ruta de costo mínimo.

2. Desarollo del Laboratorio

2.1. 8-Puzzle

Planteamiento Tablero 3×3 con 8 fichas numeradas y un hueco (0). El objetivo se solicita al usuario, tipicamente se representa como:

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$$

Modelado

- **Estado:** matriz 3×3 de 9 enteros; representación: `tuple(lista_plana)` para usar en conjuntos/diccionarios.
- **Acciones:** mover ficha arriba/abajo/izquierda/derecha (según posición del hueco).
- **Transición:** intercambio entre posición del hueco y el lugar siguiente.
- **Costo:** normalmente 1 por movimiento.

Solución No todas las permutaciones de las 9 casillas tienen solución. La condición clásica:

- Para el 8-puzzle, el estado es resoluble si la paridad del número de inversiones del estado inicial coincide con la paridad del estado meta.

Heurísticas recomendadas

- **Número de fichas fuera de lugar** (simple, débil).
- **Distancia Manhattan** (más informativa y admisible).

Implementación (puntos clave)

- **Representación:** usar `tuple` para claves en `set/dict`.
- **Generación de vecinos:** intercambiar 0 con sus vecinos válidos.
- **A*:** usar `heapq` para la frontera; almacenar $f, g, estado$; mantener `came_from` para reconstrucción.

Detalles adicionales del código El código sigue estos pasos principales:

1. Inicializa el tablero inicial y el estado meta como tuplas.
2. Implementa una función de vecinos que calcula los posibles movimientos del hueco.
3. Define la función heurística (*Manhattan*).
4. Ejecuta A* expandiendo nodos y actualizando costos g.
5. Cuando alcanza el estado meta, reconstruye el camino recorrido paso a paso.

2.2. Lámpara

Planteamiento El sistema comprende dos estados: `apagada` y `encendida`. Las acciones son `prender` y `apagar`.

Teoría añadida Es un sistema determinista; no requiere búsqueda sofisticada: si queremos encender es se escoge el estado de `prender` o si es de apagar `apagada`.

Pruebas y casos borde Validar que las acciones no rompen el estado solo en el conjunto {`apagada,encendida`}).

Detalles adicionales del código

1. Se define una función `cambiar_estado(accion)` que recibe la acción y retorna el nuevo estado.
2. Cada vez que se ejecuta, imprime el estado actual.
3. Se compara el resultado con el estado meta (`encendida`) y se indica si ya se alcanzó.

2.3. Mascota (aleatoriedad)

Planteamiento Estado emocional de la mascota depende de una entrada aleatoria (cantidad de comida). Reglas:

- Si recibe exactamente 10 → `feliz`.
- Si recibe menos → `triste`.

Implementación (puntos clave)

- Generador de aleatorios: `random.randint(1,10)` o distribución personalizada.
- Transición: simple función que devuelve nuevo estado según la regla.

Detalles adicionales del código

1. El código comienza inicializando el estado en `Neutro`.
2. Se genera un número aleatorio de comida en cada ciclo.
3. Según el valor, se asigna el estado correspondiente: `feliz` o `triste`.
4. Se imprime el resultado junto con la cantidad de comida recibida.

2.4. Búsqueda del Tesoro

Planteamiento El jugador se encuentra en la grilla 3×3 desde $(0, 0)$ hasta $(2, 2)$. Movimientos sencillos en 4 direcciones.

Se implementó una política greedy: elegir una acción que reduzca la distancia Manhattan al objetivo.

Análisis

- **Ventaja:** simple, muy rápido en un grid pequeño.
- **Inconveniente:** no garantiza solución si hay obstáculos y que deba desplazarse alrededor del mismo.

Detalles adicionales del código

1. Se representa el estado como coordenadas (x, y).
2. Se calcula en cada paso la distancia Manhattan hasta el tesoro.
3. Se genera una lista de posibles movimientos válidos.
4. Se elige el movimiento que más reduzca la distancia.
5. Se imprime el camino seguido hasta llegar a la meta.

2.5. Laberinto 3×3

Planteamiento El Jugador se encuentra en una grilla 3×3 con 3 obstáculos seleccionados aleatoriamente (sin bloquear inicio/meta). Inicio $(0, 0)$, meta $(2, 2)$.

Estructuras de datos usadas

- `heapq` con tuplas `(f, g, estado)`

Pseudocódigo A* (compacto)

1. Inicializar `open` con el estado inicial (f = h(start), g=0).
2. Mientras `open` no vacío:
   - Extraer estado con menor f.
   - Si es la meta $\rightarrow$ reconstruir camino con `came_from` & terminar.
   - Para cada vecino válido:
     - Calcular $g_{nuevo} = g(actual) + cost$ (cost=1).
     - Si vecino no está en closed y mejora g conocida, actualizar `came_from`, empujar en open con $f = g_{nuevo} + h(vecino)$.

Detalles del código

1. Se genera el laberinto en forma de matriz 3×3 con obstáculos "X".
2. Se aplica A* considerando la heurística Manhattan.
3. El jugador comienza en "J" y la meta es "M".
4. El programa imprime el laberinto paso a paso mostrando cómo se avanza.
5. Una vez encontrada la meta, se muestra la ruta completa en el mapa.