Laboratorio 1 Introducción a la Inteligencia Artificial

David Alejandro Avendaño Rodríguez, Manuel Santiago Ibañez Gutierrez

2 de septiembre de 2025

Resumen

Este informe presenta el desarrollo e implementación en Python de tres algoritmos básicos orientados a la resolución de problemas mediante técnicas de búsqueda y transición de estados. El primer algoritmo aplica búsqueda en anchura (BFS) para resolver el clásico rompecabezas del 8-Puzzle, garantizando la obtención de la solución óptima en el menor número de movimientos. El segundo algoritmo aborda dos problemas simples: el cambio de estado de una mascota de "triste" a "contenta" mediante la acción de alimentarla, y el recorrido de un pirata en una cuadrícula hasta llegar a un tesoro. Finalmente, el tercer algoritmo plantea la resolución de un laberinto con obstáculos, combinando exploración manual por parte del usuario y búsqueda automática mediante BFS cuando no se logra alcanzar la meta. Los resultados obtenidos permiten observar cómo diferentes enfoques de búsqueda se adaptan a problemas de distinta complejidad. Asimismo, se resalta la importancia de los modelos de estados y de los algoritmos de búsqueda en laconstrucción de agentes inteligentes capaces de alcanzar objetivos de manera eficiente.

1. Introducción

La Inteligencia Artificial (IA) emplea distintos enfoques para resolver problemas que involucran la toma de decisiones, la planificación y la navegación en espacios de estados. Dentro de estos enfoques, los algoritmos de búsqueda juegan un papel fundamental, ya que permiten explorar sistemáticamente posibles soluciones y seleccionar la más adecuada según los objetivos planteados.El presente laboratorio tiene como propósito introducir conceptos básicos de búsqueda y transición de estados mediante la implementación de tres algoritmos en Python. Cada uno de ellos aborda un problema diferente en cuanto a complejidad y dinámica:

- El primer algoritmo aplica búsqueda en anchura (*Breadth-First Search*, BFS) para resolver el problema del 8-Puzzle, ilustrando cómo explorar un espacio de estados hasta alcanzar una configuración objetivo de manera óptima.
- El segundo algoritmo se enfoca en dos problemas sencillos de transición de estados: el cambio de estado de una mascota de "triste" a "contenta", y el movimiento de un pirata en una cuadrícula hasta encontrar un tesoro.
- El tercer algoritmo aborda la resolución de un laberinto con obstáculos, incorporando tanto exploración manual como búsqueda automática mediante BFS, lo que permite combinar la interacción humana con la resolución sistemática de un agente.

2. Objetivos

2.1. Objetivo General

Implementar y analizar diferentes algoritmos de búsqueda y transición de estados en Python, con el fin de comprender su funcionamiento y aplicabilidad en problemas de diversa complejidad dentro del contexto de la Inteligencia Artificial.

2.2. Objetivos Específicos

- Comprender el funcionamiento del algoritmo de búsqueda en anchura (BFS) aplicado a la resolución del 8-Puzzle.
- Modelar problemas sencillos de transición de estados, como el cambio de estado de una mascota y el movimiento de un pirata hacia un tesoro.
- Implementar un algoritmo de resolución de laberintos que combine la interacción manual con búsqueda automática mediante BFS.
- Analizar los resultados obtenidos para identificar ventajas, limitaciones y posibles aplicaciones de cada algoritmo.

3. Desarrollo

3.1. Algoritmo 1: Búsqueda en Anchura para el 8-Puzzle

El primer algoritmo implementa el método de *Breadth-First Search* (BFS) para resolver el clásico problema del 8-Puzzle. Este rompecabezas consiste en una cuadrícula de 3×3 con fichas numeradas del 1 al 8 y un espacio vacío (representado con 0). El objetivo es pasar de una configuración inicial a una configuración final deseada mediante movimientos válidos que intercambian la posición del espacio vacío con fichas adyacentes. El algoritmo BFS utiliza una cola (`deque`) para explorar los estados nivel por nivel, lo que garantiza encontrar la solución más corta en cuanto al número de movimientos. Durante la ejecución, se almacenan los estados visitados para evitar ciclos y redundancias. De esta manera, se asegura que la primera vez que se alcanza el estado final, éste corresponde a la solución óptima.

```
[language=Python]
from collections import deque

# Estado final deseado
estado_final = [[1, 2, 3],
                [8, 0, 4],
                [7, 6, 5]]

# Estado inicial
estado_inicial = [[2, 8, 3],
                  [1, 6, 4],
                  [7, 0, 5]]

# Movimientos posibles: arriba, abajo, izquierda, derecha
movimientos = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Función para encontrar la posición del cero (espacio vacío)
def encontrar_vacio(tablero):
    for i in range(3):
        for j in range(3):
            if tablero[i][j] == 0:
                return i, j

# Función para convertir lista en tupla (hashable)
def a_tupla(tablero):
    return tuple(tuple(fila) for fila in tablero)

# BFS para resolver el puzzle
def bfs(estado_inicial, estado_final):
    cola = deque()
```

```
    cola.append((estado_inicial, []))  # Estado + ruta de movimientos
    visitados = set()
    visitados.add(a_tupla(estado_inicial))

    while cola:
        estado, camino = cola.popleft()

        if estado == estado_final:
            return camino

        x, y = encontrar_vacio(estado)

        for dx, dy in movimientos:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                nuevo_estado = [fila[:] for fila in estado]
                nuevo_estado[x][y], nuevo_estado[nx][ny] = nuevo_estado[nx][ny], nuevo_estado[x][y]
                if a_tupla(nuevo_estado) not in visitados:
                    visitados.add(a_tupla(nuevo_estado))
                    cola.append((nuevo_estado, camino + [nuevo_estado]))
    return None

# Ejecutar BFS
solucion = bfs(estado_inicial, estado_final)

# Mostrar la solución paso a paso
if solucion:
    print("Solución encontrada en", len(solucion), "movimientos:")
    for paso in solucion:
        for fila in paso:
            print(fila)
        print()
else:
    print("No se encontró solución.")
```

Al ejecutar el código con la configuración inicial y final propuestas, el programa muestra la secuencia de movimientos que llevan desde el estado inicial hasta el estado objetivo. El resultado incluye la cantidad mínima de pasos requeridos y la evolución del tablero paso a paso. Este enfoque permite resolver el 8-Puzzle de forma sistemática, garantizando tanto la existencia de una solución como su optimalidad en número de movimientos.

3.2. Algoritmo 2: Problemas de Transición de Estados

El segundo algoritmo aborda dos problemas sencillos que ilustran el concepto de transición de estados en un sistema. En ambos casos, se define un estado inicial, un estado objetivo y las acciones posibles para pasar de uno al otro.

- **Problema 1: Mascota triste**
  La mascota puede encontrarse en dos estados: "triste" y "contenta". El agente (usuario) puede ejecutar la acción de "dar comida", lo que provoca la transición de la mascota del estado "triste" al estado "contenta". Este ejemplo muestra cómo un agente puede reaccionar de manera determinística para alcanzar una meta definida.

- **Problema 2: Tesoro del pirata**
  El pirata inicia en la posición $(0, 0)$ dentro de una cuadrícula y su objetivo es llegar a la posición $(2, 2)$ donde se encuentra el tesoro. Para ello, dispone de las acciones "arriba", "abajo", "izquierda" y "derecha". El algoritmo decide de forma secuencial los movimientos correctos hasta alcanzar la meta, almacenando el camino recorrido.

```
[language=Python]
# SEGUNDO PUNTO LABORATORIO

# Problema 1: Mascota triste

# Defino estados y acciones, y el objetivo final
estados = ["triste", "contenta"]
acciones = ["Dar comida", "Nada"]

Estado = "triste"
meta = "contenta"

# Muestra estado inicial de la mascota

print("Estado de la mascota:", Estado)

# Ciclo para monitorear hasta alcanzar el estado objetivo

while Estado != meta:
    accion = "Dar comida"
    print("accion realizada:", accion)

    if accion == "Dar comida" :
            Estado = "contenta"
    else:
        Estado = "triste"

print("Estado actual:" , Estado)

#-------------------------------------------------------------------------------------------------------

# Problema 2: tesoro del pirata

# Defino acciones, es decir los movimientos del pirata

Acciones2 = { "Arriba":(0,1),
              "Abajo":(0,-1),
              "Izquierda":(-1,0),
              "Derecha":(1,0)}

# Estados iniciales y meta

Estado2 = (0,0)
Meta2 = (2,2)

print("posición actual del pirata:", Estado2)
camino = [Estado2]

# Algoritmo para llegar al tesoro, similar al bucle anterior

while Estado2 != Meta2:
     x2, y2 = Estado2

     if x2 < Meta2[0]:
          accion2 = "Derecha"
     elif y2 < Meta2[1]:
```

```
            accion2 = "Arriba"
        else:
            accion2 = None

        if accion2:
         dx, dy = Acciones2[accion2]
         Estado2 = (x2 + dx, y2 + dy)
         camino.append(Estado2)
         print(f"Acción: {accion2} → Nueva posición: {Estado2}")

print("tesoro encontrado!")
print("camino:", camino)
```

En el primer problema, la mascota inicia en el estado "triste" y tras ejecutar la acción "dar comida", cambia a "contenta", alcanzando el objetivo de manera inmediata. En el segundo problema, el pirata parte de la posición inicial $(0, 0)$ y mediante una secuencia de movimientos hacia la derecha y hacia arriba logra llegar a la posición $(2, 2)$. El algoritmo imprime cada movimiento realizado y el camino completo seguido por el pirata hasta encontrar el tesoro. Estos ejemplos muestran cómo los problemas pueden representarse mediante estados y acciones, facilitando la construcción de agentes reactivos y secuenciales.

3.3. Algoritmo 3: Resolución de Laberinto con Obstáculos

El tercer algoritmo plantea un entorno más complejo en el que un agente debe desplazarse en una cuadrícula desde una posición inicial $(0, 0)$ hasta una meta ubicada en $(n - 1, n - 1)$. En este entorno existen obstáculos que restringen los movimientos posibles,lo que obliga al agente a buscar rutas alternativas.

El algoritmo combina dos modos de operación:

- **Modo manual:** el usuario ingresa los movimientos (arriba, abajo, izquierda, derecha). Si el movimiento es inválido o el camino está bloqueado por un obstáculo, el agente no avanza. En caso de llegar a la meta, se muestra el camino recorrido manualmente.
- **Modo automático:** si el usuario no logra alcanzar la meta, el sistema ejecuta una búsqueda en anchura (BFS) que explora los posibles caminos hasta encontrar el más corto que lleve al agente a la meta, evitando los obstáculos.

```
[language=Python]
from collections import deque

tamaño = 3
inicio = (0, 0)
meta = (tamaño-1, tamaño-1)
obstaculos = {(1, 1)}

acciones = {
    "derecha": (0, 1),
    "izquierda": (0, -1),
    "abajo": (1, 0),
    "arriba": (-1, 0)
}

def mover(estado, accion, obstaculos=set()):
    if accion not in acciones:
        return estado
    dx, dy = acciones[accion]
    nuevo_estado = (estado[0] + dx, estado[1] + dy)
    if not (0 <= nuevo_estado[0] < tamaño and 0 <= nuevo_estado[1] < tamaño):
```

```
        return estado
    if nuevo_estado in obstaculos:
        return estado
    return nuevo_estado

def buscar_camino(inicio, meta, obstaculos):
    cola = deque([(inicio, [inicio])])
    visitados = set([inicio])
    while cola:
        estado, camino = cola.popleft()
        if estado == meta:
            return camino
        for accion in acciones:
            nuevo_estado = mover(estado, accion, obstaculos)
            if nuevo_estado not in visitados and nuevo_estado != estado:
                visitados.add(nuevo_estado)
                cola.append((nuevo_estado, camino + [nuevo_estado]))
    return None

print("=== LABERINTO MANUAL ===")
print(f"Inicio: {inicio}, Meta: {meta}, Obstáculos: {obstaculos}")
print("Movimientos posibles:", list(acciones.keys()))

estado = inicio
camino_manual = [estado]

while estado != meta:
    accion = input("Ingresa movimiento (o 'salir' para parar manual): ").strip().lower()
    if accion == "salir":
        break
    estado_nuevo = mover(estado, accion, obstaculos)
    if estado_nuevo == estado:
        print("Movimiento inválido o bloqueado")
    else:
        estado = estado_nuevo
        camino_manual.append(estado)
        print("Estado actual:", estado)

if estado == meta:
    print("\n¡Meta alcanzada manualmente! ")
    print("Camino recorrido:", camino_manual)
else:
    print("\nNo se llegó manualmente, ejecutando búsqueda automática")
    camino_auto = buscar_camino(inicio, meta, obstaculos)
    if camino_auto:
        print("Camino encontrado automáticamente:", camino_auto)
    else:
        print("No existe camino disponible ")
```

Durante la ejecución, el programa primero permite al usuario intentar resolver el laberinto de manera manual. Si logra alcanzar la meta, se imprime el camino recorrido. En caso contrario, se activa el algoritmo BFS que calcula automáticamente la ruta más corta evitando obstáculos. Este enfoque híbrido permite observar cómo la interacción humana puede complementarse con algoritmos de búsqueda, resultando en un agente capaz de adaptarse a entornos con restricciones y de garantizar la solución siempre que ésta exista.

4. Conclusiones

El desarrollo de este laboratorio permitió comprender de manera práctica cómo los algoritmos de búsqueda y la representación de problemas mediante espacios de estados constituyen la base para la construcción de agentes inteligentes. A través del uso de *Breadth-First Search* (BFS), se evidenció la capacidad de encontrar soluciones óptimas en problemas complejos como el 8-Puzzle, mientras que en ejemplos más simples, como la mascota y el pirata, se destacó la utilidad de los modelos de transición de estados para alcanzar objetivos de manera determinística. Finalmente, la resolución de un laberinto con obstáculos mostró la versatilidad de combinar interacción humana con búsqueda automática, lo cual refleja un enfoque híbrido aplicable a sistemas reales. En conjunto, estos ejercicios demuestran que la elección del algoritmo y la correcta modelación del entorno son factores determinantes para garantizar eficiencia, adaptabilidad y éxito en la solución de problemas dentro del campo de la Inteligencia Artificial.

Referencias

[1] S. Russell y P. Norvig, *Artificial Intelligence: A Modern Approach*, 4ta edición, Pearson, 2021.

[2] I. Goodfellow, Y. Bengio, y A. Courville, *Deep Learning*, MIT Press, 2016. Disponible en: https://www.deeplearningbook.org

[3] N. S. Nise, *Control Systems Engineering*, 7ma edición, Wiley, 2015.

[4] Guido van Rossum y Python Software Foundation, *Python Language Reference*, versión 3.11, 2023. Disponible en: https://www.python.org

[5] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.