INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

INGENIERIA EN SISTEMAS COMPUTACIONALES

Ejercicio de laboratorio 4

“Búsqueda informada y Templado simulado”

Presentan

GOMEZ HERNANDEZ ALAN JAVIER
HERNÁNDEZ PÉREZ JUAN MANUEL
JIMÉNEZ CRUZ DANIEL
RIVAS CARRERA DIANA LAURA

Profesor

FLORIANO GARCÍA ANDRÉS

Octubre 2023

INTRODUCCION

La búsqueda informada, también conocida como búsqueda heurística, es un enfoque utilizado en la inteligencia artificial y la informática para resolver problemas de búsqueda en un espacio de estados. Se diferencia de la búsqueda no informada, que explora el espacio de búsqueda sin información adicional, en que utiliza conocimiento o heurísticas para guiar la búsqueda hacia soluciones más prometedoras de manera más eficiente.

En la búsqueda informada, se utiliza información heurística para estimar cuán cercano está un estado dado al objetivo o la solución del problema. Esta información se utiliza para priorizar la exploración de estados que parecen más prometedores en lugar de explorar ciegamente todas las posibles opciones. El objetivo es reducir el costo computacional y el tiempo necesario para encontrar una solución.

El templado simulado, también conocido como simulated annealing en inglés, es un algoritmo de optimización estocástica que se utiliza para encontrar soluciones aproximadas a problemas de optimización combinatoria y continua. Este algoritmo se inspira en un proceso físico llamado "templado" que se utiliza en la metalurgia para enfriar gradualmente un material caliente a fin de obtener una estructura cristalina deseada y evitar defectos en el material.

El algoritmo de templado simulado comienza con una solución inicial y busca iterativamente soluciones vecinas mientras disminuye gradualmente una "temperatura" simulada. La temperatura controla la probabilidad de aceptar soluciones peores en busca de soluciones óptimas. A medida que la temperatura disminuye, la probabilidad de aceptar soluciones peores disminuye, lo que permite al algoritmo converger hacia una solución aproximadamente óptima.

DESARROLLO

1 Empleando búsqueda informada (coste uniforme, A* o una variante) resuelve alguno de los siguientes problemas:
   - Las 8 reinas.
   - 8-puzzle.
   - Laberinto de al menos 20 filas por 20 columnas

Nodo: Una clase para representar un estado del tablero, que incluirá:

- El estado actual del tablero.
- El nodo padre, que nos permitirá rastrear el camino una vez que encontremos la solución.
- g: El coste del camino desde el estado inicial al nodo actual.
- h: La heurística del nodo, calculada como la distancia Manhattan.
- f: La suma de g y h.

Movimientos posibles: Una función que generará todos los estados hijos posibles de un nodo dado.

Distancia Manhattan: Una función para calcular la distancia Manhattan de un estado dado al estado objetivo.

Algoritmo A*: Implementaremos el bucle principal del algoritmo A*, utilizando las funciones y estructuras anteriores.

En nuestro caso, probamos con el estado inicial:

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

```python
import heapq
import time
import numpy as np

# Definimos el estado objetivo del puzzle 8
TARGET = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])


class Node:
    def __init__(self, state, parent, g, h):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
        self.f = g + h

    def __lt__(self, other):
```

```
        return self.f < other.f
```

La heurística usada es la de **distancia Manhattan**. La siguiente función calcula la distancia Manhattan desde un estado dado hasta el estado objetivo. La distancia Manhattan es simplemente la suma de las distancias horizontales y verticales de cada ficha desde su posición actual hasta su posición objetivo.

```
def manhattan_distance(state):
    # Calcula la distancia Manhattan desde un estado hasta el objetivo.
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i, j] != 0:  # Ignora el espacio vacío
                # Obtiene la posición del número en el estado objetivo
                target_i, target_j = divmod(state[i, j] - 1, 3)
                # Suma la distancia Manhattan para el número actual
                distance += abs(i - target_i) + abs(j - target_j)
    return distance
```

La función get_possible_moves genera todos los movimientos posibles desde un estado dado. Encuentra la posición del espacio vacío y luego intenta moverlo en todas las direcciones posibles (arriba, abajo, izquierda, derecha), siempre que el movimiento sea válido (no salga del tablero).

```
def get_possible_moves(state):
    """Genera todos los movimientos posibles desde un estado dado."""
    possible_moves = []
    # Encuentra la posición del 0 (espacio vacío)
    i, j = np.where(state == 0)
    i, j = i[0], j[0]

    # Definimos los posibles movimientos como (di, dj) donde d es la dirección en
la que se mueve el 0
    for di, dj in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        ni, nj = i + di, j + dj
        # Verifica que los nuevos índices estén dentro de los límites del tablero
        if 0 <= ni < 3 and 0 <= nj < 3:
            new_state = state.copy()
            # Realiza el movimiento
            new_state[i, j], new_state[ni, nj] = new_state[ni, nj], new_state[i,
j]
            possible_moves.append(new_state)

    return possible_moves
```

Definimos la función a_star_search de la siguiente manera:

- Inicializa la cola de prioridad pq y el conjunto explored.

- Crea el nodo inicial y lo añade a pq.
- Mientras pq no esté vacía, extrae el nodo con el menor valor de f y lo expande, generando sus sucesores y añadiéndolos a pq si no han sido explorados previamente.
- Si se encuentra el estado objetivo, reconstruye el camino desde el estado inicial hasta el objetivo y lo retorna.
- Si pq se vacía sin encontrar una solución, retorna None.

```
def a_star_search(initial_state):
    """Implementa el algoritmo A* para encontrar la solución al puzzle 8."""
    # Usa una cola de prioridad para almacenar los nodos
    pq = []
    # Crea el nodo inicial y lo añade a la cola
    initial_node = Node(initial_state, None, 0,
manhattan_distance(initial_state))
    heapq.heappush(pq, initial_node)

    # Usa un conjunto para almacenar los estados explorados
    explored = set()

    while pq:
        # Obtiene el nodo con el menor valor de f de la cola
        current_node = heapq.heappop(pq)
        current_state = current_node.state

        # Si el estado actual es el objetivo, reconstruye el camino y lo retorna
        if np.array_equal(current_state, TARGET):
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]

        # Marca el estado actual como explorado
        explored.add(tuple(current_state.flatten()))

        # Genera los sucesores del estado actual
        for move in get_possible_moves(current_state):
            # Si el sucesor ya ha sido explorado, lo ignora
            if tuple(move.flatten()) in explored:
                continue
            # Crea un nuevo nodo y lo añade a la cola
            new_node = Node(move, current_node, current_node.g + 1,
manhattan_distance(move))
            heapq.heappush(pq, new_node)

    # Si la cola está vacía y no se ha encontrado una solución, retorna None
    return None


if __name__ == '__main__':
    # Ejemplo de estado inicial
    initial_state = np.array([[8, 1, 6], [5, 4, 7], [2, 3, 0]])

    # Tiempo inicial
    start_time = time.time()
    # Buscar la solución
```

```python
    solution_path = a_star_search(initial_state)
    # Tiempo final
    end_time = time.time()

    # Imprimir la solución
    print("Pasos para llegar al objetivo: \n")
    for state in solution_path:
        print(state, "\n")

    print(f"Solución encontrada en {len(solution_path)} pasos")
    print(f"Tiempo de ejecución: {end_time - start_time} segundos")
```

Pruebas de funcionamiento

```
Pasos para llegar al objetivo:


[[8 1 6]
 [5 4 7]
 [2 3 0]]

[[8 1 6]
 [5 4 0]
 [2 3 7]]

[[8 1 0]
 [5 4 6]
 [2 3 7]]

[[8 0 1]
 [5 4 6]
 [2 3 7]]
```

```
[[1 2 3]
 [4 0 5]
 [7 8 6]]

[[1 2 3]
 [4 5 0]
 [7 8 6]]

[[1 2 3]
 [4 5 6]
 [7 8 0]]

Solución encontrada en 25 pasos
Tiempo de ejecución: 0.07964372634887695 segundos
```

Se resuelve el problema solicitado con el ejercicio de 8-puzzle, que como se ve en captura, encuentra una solución de este estado inicial específico mediante 25 pasos y así mismo el tiempo de ejecución es el mínimo

DESARROLLO

2. Empleando templado simulado (simulated annealing) encuentra el valor mínimo de las siguientes funciones:

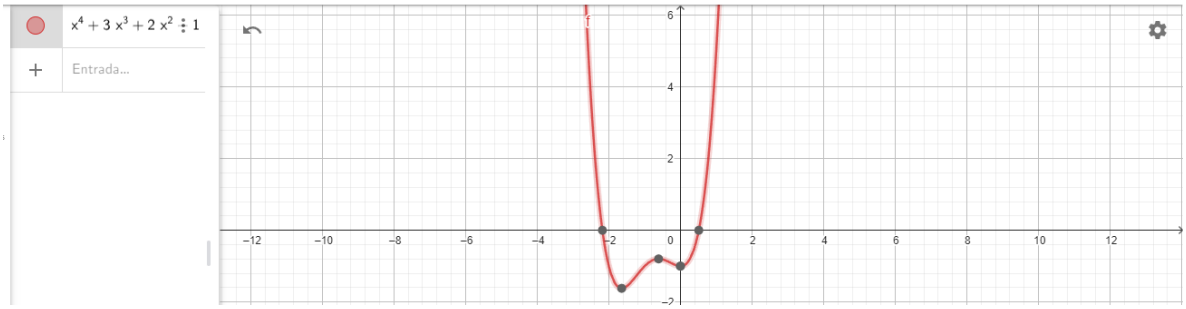
1. f(x)=x^(4)+3*x^(3)+2*x^(2)-1
2. f(x)=x^(2)-3*x-8

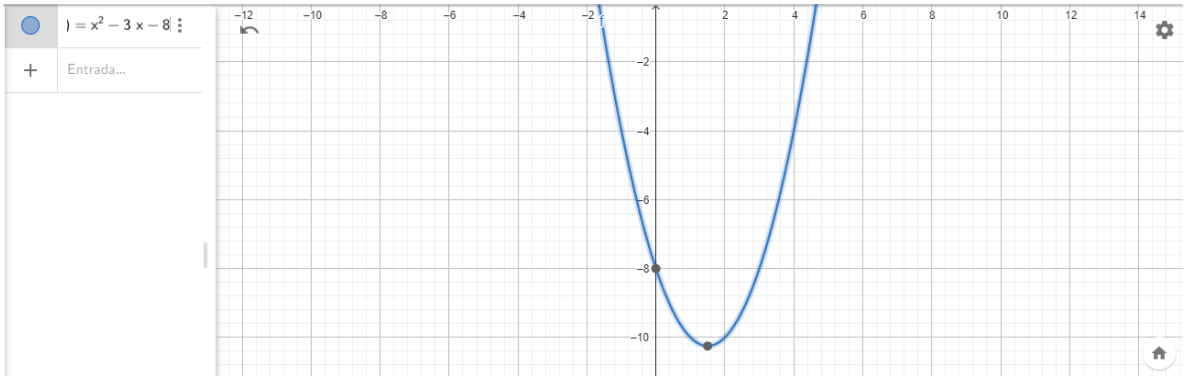
```
    import math
import random

# f(x) a minimizar
def f1(x):
    return x**4 + 3*x**3 + 2*x**2 - 1

def f2(x):
    return x**2 - 3*x - 8

# Función de recocido simulado
def simulated_annealing(func, initial_state, T, T_MIN, cooling_rate,
num_iterations):
    current_state = initial_state
    best_state = current_state

    while T > T_MIN:
        for i in range(num_iterations):
            # Estado vecino: La idea es explorar las soluciones cercanas al
estado actual
            neighbor = current_state + random.uniform(-0.1, 0.1)

            # Diferencia entre la función objetivo en el nuevo estado y el estado
actual
            delta = func(neighbor) - func(current_state)

            # Si el nuevo estado es mejor o es aceptado con una probabilidad,
actualízalo
            if delta < 0 or random.random() < math.exp(-delta / T):
                current_state = neighbor

            # Actualizar la mejor solución encontrada
            if func(current_state) < func(best_state):
                best_state = current_state

        # Enfriar la temperatura
        T *= cooling_rate

    return best_state

# Parámetros del algoritmo
initial_state = 0.0  # Estado inicial
T = 100.0            # Temperatura inicial
T_MIN = 0.01         # Temperatura mínima
cooling_rate = 0.99  # Tasa de enfriamiento
num_iterations = 100 # Iteraciones por temperatura

# Primera funcion aplicacando el recocido simulado
best_solution_f1 = simulated_annealing(f1, initial_state, T, T_MIN, cooling_rate,
num_iterations)
print("Mejor solución para f(x) = x^4 + 3x^3 + 2x^2 - 1:", best_solution_f1)
print("Valor mínimo encontrado:", f1(best_solution_f1))
```

```python
# Segunda funcion aplicacando el recocido simulado
best_solution_f2 = simulated_annealing(f2, initial_state, T, T_MIN, cooling_rate, num_iterations)
print("\nMejor solución para f(x) = x^2 - 3x - 8:", best_solution_f2)
print("Valor mínimo encontrado:", f2(best_solution_f2))
```

PRUEBAS DE FUNCIONAMIENTO

```
C:\Users\alang\anaconda3\python.exe "C:\Users\alang\Downloads\P4_recocidoSimulado (1).py"
Mejor solución para f(x) = x^4 + 3x^3 + 2x^2 - 1: -1.6403862959946054
Valor mínimo encontrado: -1.6196843494144586


Mejor solución para f(x) = x^2 - 3x - 8: 1.5000045236522752
Valor mínimo encontrado: -10.249999999979536
```





CONCLUSIÓN

A lo largo de estas prácticas, se exploraron dos algoritmos de búsqueda y optimización fundamentales en el campo de la Inteligencia Artificial: el Algoritmo A* y el Recocido Simulado. Cada uno de estos algoritmos aborda el problema de la búsqueda y la optimización de manera única, proporcionando soluciones viables para diferentes tipos de problemas.

Con el Algoritmo A*, se abordó el problema clásico del 8-puzzle, un ejemplo paradigmático de problemas de espacio de estados. A través de la implementación y ejecución del algoritmo, se pudo observar la eficacia de A* en la exploración dirigida del espacio de búsqueda, utilizando una heurística (la distancia Manhattan) para guiar la búsqueda hacia soluciones óptimas de manera eficiente. La utilización de una heurística adicional es vital en la resolución de problemas complejos y de alta dimensionalidad, proporcionando un medio para explorar soluciones potenciales de manera estratégica, minimizando el costo computacional y el tiempo necesario para encontrar una solución.

En cuanto al Recocido Simulado, se exploró su aplicación en la optimización de funciones matemáticas. A pesar de ser un algoritmo metaheurístico estocástico y, por lo tanto, no garantizar encontrar el óptimo global en cada ejecución, mostró ser una herramienta poderosa para aproximarse a soluciones óptimas en espacios de búsqueda amplios o mal definidos, especialmente cuando se carece del conocimiento preciso del paisaje del problema. La capacidad del Recocido Simulado para escapar de óptimos locales mediante la aceptación de soluciones peores (con cierta probabilidad) es fundamental para explorar de manera más exhaustiva el espacio de soluciones, brindando una flexibilidad y robustez únicas en la búsqueda de soluciones óptimas.

Estas prácticas resaltaron la importancia de seleccionar y adaptar algoritmos de búsqueda y optimización de acuerdo con las características y requisitos del problema en cuestión. Mientras que el Algoritmo A* es particularmente eficiente para problemas de búsqueda en espacio de estados con una heurística clara y admisible, el Recocido Simulado proporciona una metodología robusta y flexible para explorar espacios de soluciones en problemas de optimización complejos y altamente no lineales.