TECNOLÓGICO NACIONAL DE MÉXICO

Baez Sauceda Jesus Arnoldo

Prof. Jose Mario Rios Felix

Inteligencia Artificial

Tarea 4

17 de Septiembre de 2025

```python
from collections import deque, defaultdict
import heapq
import math


class Nodo:
    def __init__(self, estado, padre=None, accion=None,
costo=0):
        self.estado = estado
        self.padre = padre
        self.accion = accion
        self.costo = costo

    def __lt__(self, otro):
        return self.costo < otro.costo


# 1. BÚSQUEDA PRIMERO EN ANCHURA (BFS)
def busqueda_primero_anchura(estado_inicial, es_objetivo,
obtener_sucesores):
    frontera = deque([Nodo(estado_inicial)])
    visitados = set([estado_inicial])

    while frontera:
        nodo_actual = frontera.popleft()

        if es_objetivo(nodo_actual.estado):
            return reconstruir_camino(nodo_actual)

        for accion, estado_siguiente, costo in
obtener_sucesores(nodo_actual.estado):
            if estado_siguiente not in visitados:
                visitados.add(estado_siguiente)
                nuevo_nodo = Nodo(estado_siguiente,
nodo_actual, accion, nodo_actual.costo + costo)
                frontera.append(nuevo_nodo)

    return None  # No se encontró solución


# 2. BÚSQUEDA PRIMERO EN PROFUNDIDAD (DFS)
def busqueda_primero_profundidad(estado_inicial, es_objetivo,
obtener_sucesores, limite_profundidad=float('inf')):
    frontera = [Nodo(estado_inicial)]
    visitados = set([estado_inicial])

    while frontera:
```

```
        nodo_actual = frontera.pop()

        if es_objetivo(nodo_actual.estado):
            return reconstruir_camino(nodo_actual)

        if len(reconstruir_camino(nodo_actual)) <
limite_profundidad:
            for accion, estado_siguiente, costo in
reversed(list(obtener_sucesores(nodo_actual.estado))):
                if estado_siguiente not in visitados:
                    visitados.add(estado_siguiente)
                    nuevo_nodo = Nodo(estado_siguiente,
nodo_actual, accion, nodo_actual.costo + costo)
                    frontera.append(nuevo_nodo)

    return None


# 3. BÚSQUEDA COSTO UNIFORME (UCS)
def busqueda_costo_uniforme(estado_inicial, es_objetivo,
obtener_sucesores):
    frontera = []
    heapq.heappush(frontera, (0, Nodo(estado_inicial)))
    visitados = set()
    costos = {estado_inicial: 0}

    while frontera:
        costo_actual, nodo_actual = heapq.heappop(frontera)

        if es_objetivo(nodo_actual.estado):
            return reconstruir_camino(nodo_actual)

        if nodo_actual.estado in visitados:
            continue

        visitados.add(nodo_actual.estado)

        for accion, estado_siguiente, costo in
obtener_sucesores(nodo_actual.estado):
            nuevo_costo = nodo_actual.costo + costo
            if estado_siguiente not in costos or nuevo_costo
< costos[estado_siguiente]:
                costos[estado_siguiente] = nuevo_costo
                nuevo_nodo = Nodo(estado_siguiente,
nodo_actual, accion, nuevo_costo)
                heapq.heappush(frontera, (nuevo_costo,
nuevo_nodo))
```

```
    return None


# 4. BÚSQUEDA BIDIRECCIONAL (Opcional)
def busqueda_bidireccional(estado_inicial, estado_objetivo,
obtener_sucesores):
    # Implementación simplificada
    frontera_inicio = deque([Nodo(estado_inicial)])
    frontera_fin = deque([Nodo(estado_objetivo)])
    visitados_inicio = {estado_inicial: None}
    visitados_fin = {estado_objetivo: None}

    while frontera_inicio and frontera_fin:
        # Expansión desde el inicio
        nodo_actual = frontera_inicio.popleft()
        for accion, estado_siguiente, costo in
obtener_sucesores(nodo_actual.estado):
            if estado_siguiente in visitados_fin:
                # Se encontró intersección
                camino_inicio =
reconstruir_camino(nodo_actual)
                camino_fin =
reconstruir_camino(visitados_fin[estado_siguiente])[::-1]
                return camino_inicio + camino_fin[1:]
            if estado_siguiente not in visitados_inicio:
                visitados_inicio[estado_siguiente] =
nodo_actual
                frontera_inicio.append(Nodo(estado_siguiente,
nodo_actual, accion))

    return None


# 5. BÚSQUEDA ITERATIVA EN PROFUNDIDAD (Opcional)
def busqueda_iterativa_profundidad(estado_inicial,
es_objetivo, obtener_sucesores, limite_maximo=100):
    for profundidad in range(limite_maximo):
        resultado =
busqueda_primero_profundidad(estado_inicial, es_objetivo,
obtener_sucesores, profundidad)
        if resultado:
            return resultado
    return None


# Función auxiliar para reconstruir el camino
```

```
def reconstruir_camino(nodo):
    camino = []
    while nodo:
        if nodo.accion:
            camino.append((nodo.accion, nodo.estado,
nodo.costo))
        nodo = nodo.padre
    return camino[::-1]


# FUNCIÓN HEURÍSTICA PROPIA (Distancia Manhattan para
problemas de grid)
def heuristica_manhattan(estado_actual, estado_objetivo):
    """Heurística de distancia Manhattan para problemas en
grid 2D"""
    if isinstance(estado_actual, tuple) and
isinstance(estado_objetivo, tuple):
        x1, y1 = estado_actual
        x2, y2 = estado_objetivo
        return abs(x1 - x2) + abs(y1 - y2)
    return 0


# TABLA COMPARATIVA DE COMPLEJIDAD
tabla_complejidad = {
    'Búsqueda Primero en Anchura (BFS)': {
        'Tiempo': 'O(b^d)',
        'Espacio': 'O(b^d)',
        'Completo': 'Sí',
        'Óptimo': 'Sí (para costos uniformes)'
    },
    'Búsqueda Primero en Profundidad (DFS)': {
        'Tiempo': 'O(b^m)',
        'Espacio': 'O(b*m)',
        'Completo': 'No (en espacios infinitos)',
        'Óptimo': 'No'
    },
    'Búsqueda Costo Uniforme (UCS)': {
        'Tiempo': 'O(b^(1 + C*/ε))',
        'Espacio': 'O(b^(1 + C*/ε))',
        'Completo': 'Sí',
        'Óptimo': 'Sí'
    },
    'Búsqueda Bidireccional': {
        'Tiempo': 'O(b^(d/2))',
        'Espacio': 'O(b^(d/2))',
        'Completo': 'Sí',
```

```
        'Óptimo': 'Depende de la búsqueda base'
    },
    'Búsqueda Iterativa en Profundidad': {
        'Tiempo': 'O(b^d)',
        'Espacio': 'O(b*d)',
        'Completo': 'Sí',
        'Óptimo': 'Sí (para costos uniformes)'
    }
}


# EJEMPLO DE USO
def ejemplo_problema():
    """Ejemplo de un problema de grid simple"""

    def es_objetivo(estado):
        return estado == (3, 3)

    def obtener_sucesores(estado):
        x, y = estado
        sucesores = []
        # Movimientos: arriba, derecha, abajo, izquierda
        movimientos = [('arriba', (x, y - 1), 1), ('derecha',
(x + 1, y), 1),
                       ('abajo', (x, y + 1), 1),
('izquierda', (x - 1, y), 1)]

        for accion, (nx, ny), costo in movimientos:
            if 0 <= nx <= 5 and 0 <= ny <= 5:  # Grid 6x6
                sucesores.append((accion, (nx, ny), costo))
        return sucesores

    return (0, 0), es_objetivo, obtener_sucesores


# Función para mostrar la tabla comparativa
def mostrar_tabla_comparativa():
    print("TABLA COMPARATIVA DE ALGORITMOS DE BÚSQUEDA")
    print("=" * 80)
    print(f"{'Algoritmo':<35} {'Tiempo':<15} {'Espacio':<15}
{'Completo':<10} {'Óptimo':<10}")
    print("-" * 80)

    for algoritmo, datos in tabla_complejidad.items():
        print(
            f"{algoritmo:<35} {datos['Tiempo']:<15}
{datos['Espacio']:<15} {datos['Completo']:<10}
```

```
{datos['Óptimo']:<10}")


# Prueba de los algoritmos
if __name__ == "__main__":
    # Mostrar tabla comparativa
    mostrar_tabla_comparativa()
    print("\n" + "=" * 80)

    # Probar con ejemplo
    estado_inicial, es_objetivo, obtener_sucesores =
ejemplo_problema()

    print(f"\nProbando algoritmos desde {estado_inicial}
hasta (3,3)")

    # Probar BFS
    print("\n1. Búsqueda Primero en Anchura:")
    camino_bfs = busqueda_primero_anchura(estado_inicial,
es_objetivo, obtener_sucesores)
    if camino_bfs:
        print(f"Camino encontrado ({len(camino_bfs)}
pasos):")
        for accion, estado, costo in camino_bfs:
            print(f"  {accion} -> {estado} (costo acumulado:
{costo})")
    else:
        print("No se encontró solución")

    # Probar UCS
    print("\n2. Búsqueda Costo Uniforme:")
    camino_ucs = busqueda_costo_uniforme(estado_inicial,
es_objetivo, obtener_sucesores)
    if camino_ucs:
        print(f"Camino encontrado ({len(camino_ucs)} pasos,
costo total: {camino_ucs[-1][2]}):")
        for accion, estado, costo in camino_ucs:
            print(f"  {accion} -> {estado} (costo acumulado:
{costo})")

    # Probar heurística
    print(f"\n3. Heurística Manhattan de (0,0) a (3,3):
{heuristica_manhattan((0, 0), (3, 3))}")
```