Trabajo Práctico 1

Algoritmos de Busqueda en Torres de Hanoi

Abril Noguera - Pablo Brahim - Fermin Rodriguez - Kevin Pennington

En clase presentamos el problema de la torre de Hanoi. Además, vimos diferentes algoritmos de búsqueda que nos permitieron resolver este problema. Para este trabajo práctico, deberán implementar un método de búsqueda para resolver con 5 discos, del estado inicial y objetivo que se observa en la siguiente imagen:

Link al Repositorio

Adjunto el link al repositorio con la resulución completa. Repositorio de Github

---

1. ¿Cuáles son los PEAS de este problema? (Performance, Environment, Actuators, Sensors)

Supuesto del Problema

El problema de las **Torres de Hanoi** consiste en trasladar un conjunto de **5 discos** desde una **varilla inicial** a una **varilla final**, utilizando una **varilla auxiliar**, siguiendo reglas estrictas:

1. Solo se puede mover un disco a la vez.
2. Cada movimiento translada un disco de una varilla a otra.
3. Un disco no puede colocarse sobre otro más pequeño.
4. El objetivo es mover todos los discos a la última varilla en la menor cantidad de movimientos posible.

Para esta simulación en **Python**, consideramos un **agente computacional** que toma decisiones y ejecuta movimientos basados en un algoritmo de búsqueda, asegurando que los movimientos sean válidos y eficientes.

El **estado inicial** es un conjunto de 5 discos apilados en la primera varilla, mientras que el **estado final** se alcanza cuando todos los discos están en la tercera varilla, en el mismo orden.

Especificación del Entorno (PEAS)

Categoría	Descripción
Performance	- Minimizar la cantidad de movimientos. - Cumplir con las reglas del juego. - Resolver el problema en el menor tiempo computacional posible. - Resolver el problema en el menor costo computacional (memoria) posible.
Environment	- Tres varillas representadas como listas. - Cinco discos de diferentes tamaños, representados como enteros segun su tamaño.
Actuators	- Mover un disco de una varilla a otra, validando reglas. - Ejecutar movimientos en `ActionHanoi`.
Sensors	- Estado actual de las varillas y posición de los discos. - Validación de movimientos permitidos. - Número de discos que no estan en la varilla objetivo.

---

2. ¿Cuáles son las propiedades del entorno de trabajo?

Propiedades del entorno

El problema de la Torre de Hanoi se resuelve en un entorno:

- **Observable:** Ves todos los discos en todo momento.
- **Determinista:** Cada movimiento tiene un resultado predecible, no hay aleatoriedad.
- **Secuencial:** Requiere una serie de movimientos planificados (no es un solo paso).
- **Estático:** El entorno no cambia solo, necesitan que un actor los mueva.
- **Discreto:** Estados y acciones claros (mover un disco de un poste a otro).
- **Agente individual:** Solo un actor, no hay oponentes ni cooperación.

---

3. En el contexto de este problema, establezca cuáles son los: estado, espacio de estados, árbol de búsqueda, nodo de búsqueda, objetivo, acción y frontera.

- **Estado:** El estado representa cómo estan distribuidos los discos en los tres postes en un momento determinado, osea, en qué torre esta cada disco y en

qué orden.

- **Espacio de estados:** El espacio de estados es el conjunto de todas las configuraciones posibles de los discos en todas las torres.
- **Árbol de búsqueda:** El árbol de búsqueda es la estructura que representa todas las posibles secuencias de movimientos desde el estado inicial hasta el estado objetivo. En este caso, cada nodo es una configuración de discos, cada rama es una acción que une un estado con otro (haciendo movimientos legales)
- **Nodo de búsqueda:** Un nódo de busqueda es un estado específico del árbol donde también sabemos el nodo del que vino, la acción que llevo a ese estado y la profunidad del árbol.
- **Objetivo:** El objetivo en este contexto es llegar a un estado en donde todos los discos esten en la torre destino y en el orden correcto (del más grande abajo al más chico arriba de todo)
- **Acción:** Una acción es mover el disco superior de una torre a otra, cumpliendo con las reglas del problema.
- **Frontera:** La frontera es el conjunto de nodos generados pero aún no explorados en el arbol de búsqueda, osea los siguientes candidatos a ser expandidos (dependiendo el algoritmo utilizado)

---

4. Implemente algún método de búsqueda. Puedes elegir cualquiera menos búsqueda en anchura primero (el desarrollado en clase). Sos libre de elegir cualquiera de los vistos en clases, o inclusive buscar nuevos.

Algoritmo de Busqueda A*

In [6]:
```
from aima_libs.hanoi_states import ProblemHanoi, StatesHanoi
from aima_libs.tree_hanoi import NodeHanoi
from aima_libs.aima import PriorityQueue
```

Función Heurística

1. Heurística Basada en Discos Correctos

Esta heurística cuenta cuántos discos están correctamente colocados en la torre final, desde arriba hacia abajo. Al encontrar discos mal ubicados, deja de contar, y estima el número restante de discos por colocar correctamente.

- Ventaja: heurística simple, consistente y rápida.
- Desventaja: puede ser poco informativa en etapas tempranas, ya que solo observa la parte correcta del estado final.

In [17]:
```python
def heuristic_correct(node, goal):
    state = node.state
    state_rod = state.rods[2]
    goal_rod = goal.rods[2]
    count = 0

    # Cuenta cuántos discos están en la posición correcta
    for a, b in zip(state_rod, goal_rod):
        print(f"{a} {b}")
        if a == b:
            print("ok")
            count += 1
        else:
            break

    return len(goal_rod) - count
```

2. Heurística Basada en Discos Fuera de Lugar

Esta heurística penaliza: - Cada disco que aún está en las torres intermedias o inicial. - El desorden en la torre final, sumando un costo adicional si los discos colocados allí no siguen el orden esperado del objetivo.

- Ventaja: más sensible al estado global, detecta desorden y discos pendientes. Se obtiene una estimación más informada del número de movimientos restantes.
- Desventaja: más costosa computacionalmente, pero sigue siendo eficiente.

In [11]:
```python
def heuristic_incorrect(node, goal):
    state = node.state
    state_rod = state.rods[2]
    goal_rod = goal.rods[2]
    h = 0

    # Penaliza cada disco que no está en la torre final
    for rod_index in [0, 1]:
        h += len(state.rods[rod_index])

    # Penaliza si hay discos mal ubicados en la torre final
    for i, (actual, expected) in enumerate(zip(state_rod, goal_rod)):
        if actual != expected:
            h += len(goal_rod) - i  # Penaliza por cuántos discos están mal
            break

    return h
```

Algoritmo de Busqueda

In [13]:
```python
def astar_search(heuristic_func, number_disks=5):
    # Inicializamos el problema
    list_disks = [i for i in range(number_disks, 0, -1)]
```

```
    initial_state = StatesHanoi(list_disks, [], [], max_disks=number_disks)
    goal_state = StatesHanoi([], [], list_disks, max_disks=number_disks)
    problem = ProblemHanoi(initial=initial_state, goal=goal_state)

    def f(node):
        return node.path_cost + heuristic_func(node, goal_state)

    # Creamos el set con estados ya visitados
    explored = set()
    node = NodeHanoi(problem.initial)
    frontier = PriorityQueue(f=f, order='min')

    frontier.append(node)

    node_explored = 0

    # Creamos una cola de prioridad con el nodo inicial
    if problem.goal_test(node.state):
        metrics = {
                "solution_found": True,
                "nodes_explored": node_explored,
                "states_visited": len(explored),
                "nodes_in_frontier": len(frontier),
                "max_depth": node.depth,
                "cost_total": node.state.accumulated_cost
            }
        return node, metrics

    while len(frontier) != 0:
        node = frontier.pop()[1]
        node_explored += 1

        # Agregamos el estado del nodo al set. Esto evita guardar duplicados
        explored.add(node.state)

        if problem.goal_test(node.state):  # Comprobamos si hemos alcanzado
            metrics = {
                "solution_found": True,
                "nodes_explored": node_explored,
                "states_visited": len(explored),
                "nodes_in_frontier": len(frontier),
                "max_depth": node.depth,
                "cost_total": node.state.accumulated_cost
            }
            return node, metrics

        # Agregamos a la cola todos los nodos sucesores del nodo actual
        for next_node in node.expand(problem):
            # Solo si el estado del nodo no fue explorado
            if next_node.state not in explored:
                frontier.append(next_node)
            elif next_node in frontier:
                # Si ya está en la frontera, actualizamos su costo si es mer
                if f(next_node) < frontier[next_node]:
                    del frontier[next_node]
                    frontier.append(next_node)
```

```
    # Si no se encontro la solución, devolvemos la métricas igual
    metrics = {
        "solution_found": False,
        "nodes_explored": node_explored,
        "states_visited": len(explored),
        "nodes_in_frontier": len(frontier),
        "max_depth": node.depth, # OBS: Si no se encontró la solución, este
        "cost_total": None
    }
    return None, metrics
```

Resultados

In [ ]:
```
heuristics = [heuristic_correct, heuristic_incorrect]
for h in heuristics:
    print(f"Ejecutando A* con la heurística {h.__name__}")
    solution, metrics = astar_search(h)
    if solution:
        print(f"Solución encontrada: {solution.state.rods}")
        solution.generate_solution_for_simulator(initial_state_file=f"./init
                                                 sequence_file=f"./sequence_{h.__name

        print(f"** Ejecutar: python3 simulation_hanoi.py initial_state_{h.__

    else:
        print("No se encontró solución")
    print(f"Métricas:")
    for key, value in metrics.items():
        print(f"    - {key}: {value}")
    print("-" * 40)
```

```
Ejecutando A* con la heurística heuristic_correct
Solución encontrada: [[], [], [5, 4, 3, 2, 1]]
** Ejecutar: python3 simulator.py ./initial_state_heuristic_correct.json ./s
equence_heuristic_correct.json
Métricas:
    - solution_found: True
    - nodes_explored: 268
    - states_visited: 169
    - nodes_in_frontier: 18
    - max_depth: 31
    - cost_total: 31.0
----------------------------------------
Ejecutando A* con la heurística heuristic_incorrect
Solución encontrada: [[], [], [5, 4, 3, 2, 1]]
** Ejecutar: python3 simulator.py ./initial_state_heuristic_incorrect.json
./sequence_heuristic_incorrect.json
Métricas:
    - solution_found: True
    - nodes_explored: 222
    - states_visited: 146
    - nodes_in_frontier: 19
    - max_depth: 31
    - cost_total: 31.0
----------------------------------------
```

Execute:

`poetry shell`

`cd TP1/simulator`

`python3 simulation_hanoi.py`

- Ambas heurísticas encontraron la solución óptima, costo total de 31 movimientos ($2^k - 1$).
- La `heuristic_incorrect` necesitó menos nodos explorados y menos estados visitados, lo que sugiere una mejor orientación hacia el objetivo.

---

5. ¿Qué complejidad en tiempo y memoria tiene el algoritmo elegido?

La complejidad en tiempo de `A* search` es

$$\begin{cases} O(b \cdot d), & (\text{Mejor caso}) \\ O(b^d), & (\text{Avg}) \\ O(b^d), & (\text{Peor caso}) \end{cases}$$

[George T. Heineman, Gary Pollice, Stanley Selkow, *Algorithms in a Nutshell*, 2008]

donde b es el *branching factor* o máximo número de sucesores de cada nodo ($b = 2$ para nuestro problema) y d es la profundidad de la solución hallada ($2^k - 1$)

La complejidad en memoria de `A* search` es también (peor caso) $O(b^d)$, ya que es necesario mantener en memoria todos los nodos explorados para poder reevaluar las métricas. [Russel & Norvig, AIMA]

---

6. A nivel implementación, ¿qué tiempo y memoria ocupa el algoritmo? (Se recomienda correr 10 veces y calcular promedio y desvío estándar de las métricas).

In [24]:
```
# %pip install memory_profiler --quiet
```

In [20]:
```
import time
import statistics
from memory_profiler import memory_usage
```

In [21]:
```python
def run_trial(heuristic_func):
    start_time = time.time()

    mem_usage, (node, metrics) = memory_usage(
        (astar_search, (heuristic_func,), {'number_disks': 5}),
        retval=True,
        max_usage=True
    )

    elapsed_time = time.time() - start_time
    return elapsed_time, mem_usage, metrics
```

In [23]:
```python
heuristics = [heuristic_correct, heuristic_incorrect]
for h in heuristics:
    print(f"Resultados A* con la heurística {h.__name__}")

    # Ejecutar 10 veces
    times = []
    mems = []
    nodes_explored = []
    states_visited = []

    for _ in range(10):
        t, mem, m = run_trial(h)
        times.append(t)
        mems.append(mem)
        nodes_explored.append(m['nodes_explored'])
        states_visited.append(m['states_visited'])

    # Mostrar resultados
    print(f"Tiempo promedio: {statistics.mean(times):.4f}s ± {statistics.stc
    print(f"Memoria promedio: {statistics.mean(mems):.2f} MiB ± {statistics.
    print(f"Nodos explorados: {statistics.mean(nodes_explored):.2f} ± {stati
    print(f"Estados visitados: {statistics.mean(states_visited):.2f} ± {stat
    print("-" * 40)
```

```
Resultados A* con la heurística heuristic_correct
Tiempo promedio: 0.7881s ± 0.0725s
Memoria promedio: 51.87 MiB ± 11.62 MiB
Nodos explorados: 268.00 ± 0.00
Estados visitados: 169.00 ± 0.00
----------------------------------------
Resultados A* con la heurística heuristic_incorrect
Tiempo promedio: 0.7945s ± 0.0136s
Memoria promedio: 57.40 MiB ± 0.02 MiB
Nodos explorados: 222.00 ± 0.00
Estados visitados: 146.00 ± 0.00
----------------------------------------
```

- `heuristic_incorrect` fue más eficiente en exploración, ya que requirió menos nodos y estados para alcanzar la solución óptima.
- El tiempo de ejecución fue muy similar entre ambas heurísticas, con diferencias menores al 1%, aunque heuristic_correct presentó una mayor

variabilidad.
- La memoria utilizada fue ligeramente mayor en `heuristic_incorrect`, lo que podría explicarse por estructuras internas más complejas o mayor retención temporal de estados.

Aunque ambas heurísticas permiten encontrar la **solución óptima**, `heuristic_incorrect` resulta más informativa y reduce el número de nodos explorados, lo que la convierte en una mejor opción cuando se busca optimizar el rendimiento del algoritmo. Su pequeño costo adicional en memoria es compensado por una mejor orientación de la búsqueda hacia el objetivo.

---

7. Si la solución óptima es $2^k - 1$ movimientos con k igual al número de discos. Qué tan lejos está la solución del algoritmo implementado de esta solución óptima (se recomienda correr al menos 10 veces y usar el promedio de trayecto usado).

Distancia a la solución óptima

`heuristic_correct`: 31 - ($2^5 - 1$) = 0

`heuristic_incorrect`: 31 - ($2^5 - 1$) = 0

El algoritmo halla la solución óptima en ambos casos.

Link al Notebook

Se puede encontrar el trabajo completo en el siguiente link: Repositorio GitHub