Trabajo practico 1 - Solucion

March 24, 2024

1 TP1: Algoritmos de búsqueda en Torre de Hanoi - Solución

1.1 Tareas y preguntas a resolver:

1.1.1 Ejericio 1:

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

El entorno de trabajo (lo que se denomina PEAS) para este caso es el siguiente (se plantea una visión donde se consideraría el problema en el mundo real y se lo modela en una simulación computacional):

- Desempeño (*Performance*): En este problema, una medida de desempeño se puede basar en el la complejidad en tiempo como en espacio. O sea, cuando demora el algoritmo elejido y cuanda memoria ocupa. Otra medida posible es si es completo, o sea, si llega a una solución o no. En resumen, se pueden seleccionar medidas relacionadas con las siguientes dimensiones:
  - Completitud
  - Optimización
  - Complejidad en tiempo
  - Complejidad en espacio
- Entorno (*Environment*): El entorno de trabajo es simulado, que representa un modelo del mundo real. En este caso, en el modelo se representan las torres de Hanoi, los discos y los movimientos posibles. Todos los entes tienen su representación en números naturales.
- Actuadores (*Actuactors*): En este entorno simulado, los actuadores serían las entradas del usuario (en este caso, la consola).
- Sensores (*Sensors*): En este entorno simulado, los sensores serían la pantalla de visualización.

1.1.2 Ejericio 2:

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

Las propiedades del ambiente de trabajo para este problema se especifica en las siguientes dimensiónes como:

- **Totalmente observable** (vs "parcialmente observable"): El grafo de todas las soluciones es totalmente observable (contando con recursos infinitos, ya que dada las restricciones de memoria o tiempo, puede ser considerado como "parcialmente observable").
- **Determinista** (vs "estocástico"): Es determinista, ya que no se involucra el azar en la generación de nodos del arbol de solución.
- **Secuencial** (vs "episódico"): Un estado depende de otro, en forma de secuencia.
- **Estático** (vs "dinámico"): No se cambian los estados por acciones externas o internas.
- **Discreto** (vs "continuo"): El espacio de soluciones pertenece al conjunto de los Naturales *(Sea S el espacio de soluciones $ S N$).*
- **Agente individual** (vs "multiagente"): No participan otros agentes en el modelo.

1.1.3 Ejercicio 3:

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.

Definicón del problema	**Descripción**	**Representación**
Estado	Un estado se representa con una tupla de tuplas, ordenada de derecha a izquierda, donde cada posición de la tupa exterior representa una torre y cada valor de la tupla exterior (una tupla en sí), representa el conjunto de discos que está, en ese momento, en dicha torre. Los números dentro de la tupla interior representan a los discos.	$[posicin_1, ..., posicin_n]$ donde posiciónn es $[1, ..., m]$ discos; $\forall m, n \in N$. Ej: $[[3, 2, 1], [4], [5]]$, representa un estado en donde el disco 3, 2 , 1 se encuentran en la torre 1 (en ese orden), el disco 4 en la torre 2 y el disco 5 en la torre 3.
Espacio de estados	Todas las combinaciones posibles, siendo el número de torres elevado al número de discos.	n^m donde n representa el número de torres y m representa el número de discos, $\forall n, m \in N$
Arbol de búsqueda	Indica un camino posible, luego de aplicar la función de transición. Es una estructura de forma de arbol en donde se guardan los nodos y su posible camino.	Imagen de arbol de busqueda

Definicón del problema	Descripción	Representación
Nodo de búsqueda	Representa varios componentes, entre los cuales están, su antecesor, sus hijos y el estado actual. También se incluye el costo de llegar a dicho nodo.	Un nodo con los siguientes el siguiente estado y comportamiento: class Node: def **init**(self, state, parent=None, action=None, path_cost=0): self.state = state self.parent = parent self.action = action self.path_cost = path_cost self.depth = 0 if parent: self.depth = parent.depth + 1 def expand(self, problem): return [self.child_node(problem, action) for action in problem.actions(self.state)] def child_node(self, problem, action): next_state = problem.result(self.state, action) next_node = Node(next_state, self, action, problem.path_cost(self.path_cost, self.state, action, next_state)) return next_node def solution(self): return [node.action for node in self.path()[1:]] def path(self): node, path_back = self, [] while node: path_back.append(node) node = node.parent return list(reversed(path_back))
Objetivo	Representa una tupla objetivo, que contiene el conjunto de discos (la tupla interior) como estado final.	Un estado objetivo. Ej: $[[], [5, 4, 3, 2], [1]]$, un estado final objetivo con el primer disco en la posición 3 y la siguiente secuencia en la posición 2: disco 2, disco 3, disco 4, disco 5; para un caso de 5 discos con tres torres.

Definicón del problema	Descripción	Representación
Acción	Se interpreta como mover un disco de una torre a otra. En la representación actual, es mover la posición en la tupla de un item de la tupla interna.	Sea $f(E, D) \rightarrow E$, en donde E es un estado y D es un disco a mover.
Frontera	La función de transición que devuelve todos los estados en los cuales para pasar un disco de una torre a otra, solamente es posible si el disco actual es menor que el disco que se encuentra en la última posición de la torre objetivo.	Sea $f(E, D) -> E_1, ..., E_n$, en donde $E_1, ..., E_n$ es un conjunto de estados válidos y D es un disco a mover. Solamente son posibles estados donde el disco a mover es menor al último disco en la torre objetivo.

1.1.4 Ejercicio 4:

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.

[1]:
```
from collections import deque # Estructura de listas enlazadas para representar␣
↪colas y pilas
from hanoi_tower import tree_hanoi, search, main, hanoi_states, aima

# Siempre los importo por las dudas, capaz ni los uso...
import pandas as pd
```

1.1.5 Algoritmos de búsquedas

Algoritmos de busqueda no informada

Algoritmo busqueda en anchura (realizado en clase)

[2]:
```
def breadth_first_tree_search(problem: hanoi_states.ProblemHanoi):
    """
    Realiza una búsqueda en anchura para encontrar una solución a un problema␣
↪de Hanoi.
    Esta función no chequea si un estado se visito, por lo que puede entrar en␣
↪Loop infinitos muy fácilmente. No
    usarla con más de 3 discos.

    Parameters:
        problem (hanoi_states.ProblemHanoi): El problema de la Torre de Hanoi a␣
↪resolver.

    Returns:
```

```
        tree_hanoi.NodeHanoi: El nodo que contiene la solución encontrada.
    """
    frontier = deque([tree_hanoi.NodeHanoi(problem.initial)])  # Creamos una␣
↪cola FIFO con el nodo inicial
    while frontier:
        node = frontier.popleft()  # Extraemos el primer nodo de la cola
        if problem.goal_test(node.state):  # Comprobamos si hemos alcanzado el␣
↪estado objetivo
            return node
        frontier.extend(node.expand(problem))  # Agregamos a la cola todos los␣
↪nodos sucesores del nodo actual

    return None


def breadth_first_graph_search(problem: hanoi_states.ProblemHanoi, display:␣
↪bool = False):
    """
    Realiza una búsqueda en anchura para encontrar una solución a un problema␣
↪de Hanoi. Pero ahora si recuerda si ya
    paso por un estado e ignora seguir buscando en ese nodo para evitar␣
↪recursividad.

    Parameters:
        problem (hanoi_states.ProblemHanoi): El problema de la Torre de Hanoi a␣
↪resolver.
        display (bool, optional): Muestra un mensaje de cuantos caminos se␣
↪expandieron y cuantos quedaron sin expandir.
                                  Por defecto es False.

    Returns:
        tree_hanoi.NodeHanoi: El nodo que contiene la solución encontrada.
    """

    frontier = deque([tree_hanoi.NodeHanoi(problem.initial)])  # Creamos una␣
↪cola FIFO con el nodo inicial

    explored = set()  # Este set nos permite ver si ya exploramos un estado␣
↪para evitar repetir indefinidamente
    while frontier:
        node = frontier.popleft()  # Extraemos el primer nodo de la cola

        # Agregamos nodo al set. Esto evita guardar duplicados, porque set␣
↪nunca tiene elementos repetidos, esto sirve
        # porque heredamos el método __eq__ en tree_hanoi.NodeHanoi de aima.Node
        explored.add(node.state)
```

```
        if problem.goal_test(node.state):  # Comprobamos si hemos alcanzado el␣
↪estado objetivo
            if display:
                print(len(explored), "caminos se expandieron y", len(frontier),␣
↪"caminos quedaron en la frontera")
            return node
        # Agregamos a la cola todos los nodos sucesores del nodo actual que no␣
↪haya visitados
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and child not in␣
↪frontier)

    return None
```

Algoritmo búsqueda en profundiadad

[3]:
```
def depth_first_graph_search(problem: hanoi_states.ProblemHanoi, display: bool␣
↪= False):
    """
    Realiza una búsqueda en profundidad para encontrar una solución a un␣
↪problema de Hanoi. Recuerda los nodos para evitar la recursividad.

    Parameters:
        problem (hanoi_states.ProblemHanoi): El problema de la Torre de Hanoi a␣
↪resolver.
        display (bool, optional): Muestra un mensaje de cuantos caminos se␣
↪expandieron y cuantos quedaron sin expandir.
                                  Por defecto es False.

    Returns:
        tree_hanoi.NodeHanoi: El nodo que contiene la solución encontrada.
    """

    frontier = deque([tree_hanoi.NodeHanoi(problem.initial)])  # Creamos una␣
↪pila LIFO con el nodo inicial

    explored = set()
    while frontier:
        node = frontier.pop()  # Extraemos el último nodo de la pila

        explored.add(node.state)

        if problem.goal_test(node.state):
            if display:
                print(len(explored), "caminos se expandieron y", len(frontier),␣
↪"caminos quedaron en la frontera")
```

```
            return node
        # Agregamos a la pila todos los nodos sucesores del nodo actual que no␣
↪hayan sido visitados
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and child not in␣
↪frontier)

    return None
```

Algoritmo búsqueda en profundidad limitada iterativa

[4]:
```
def depth_limited_search(problem: hanoi_states.ProblemHanoi, limit: int = 1,␣
↪display: bool = False):
    """
    Realiza una búsqueda en profundidad para encontrar una solución a un␣
↪problema de Hanoi. Recuerda los nodos para evitar la recursividad.
    Está limitada a una profundiad dada.

    Parameters:
        problem (hanoi_states.ProblemHanoi): El problema de la Torre de Hanoi a␣
↪resolver.
        limit (int): Limite de profundidad.
        display (bool, optional): Muestra un mensaje de cuantos caminos se␣
↪expandieron y cuantos quedaron sin expandir.
                                  Por defecto es False.

    Returns:
        tree_hanoi.NodeHanoi: El nodo que contiene la solución encontrada.
    """

    frontier = deque([tree_hanoi.NodeHanoi(problem.initial)])

    explored = set()
    while frontier:
        node = frontier.pop()

        explored.add(node.state)

        if problem.goal_test(node.state):
            if display:
                print(len(explored), "caminos se expandieron y", len(
                    frontier), "caminos quedaron en la frontera")
            return node
        # Agregamos a la pila todos los nodos sucesores del nodo actual que no␣
↪hayan sido visitados, pero también inlcuimos el límite.
        frontier.extend(child for child in node.expand(problem)
```

```
                        if child.state not in explored and child not in␣
↪frontier and child.depth <= limit)

    return None


def depth_iterative_limited_search(problem: hanoi_states.ProblemHanoi, display:␣
↪bool = False):
    """
    Realiza una búsqueda en profundidad para encontrar una solución a un␣
↪problema de Hanoi. Recuerda los nodos para evitar la recursividad.
    Está limitada a una profundiad dada, sin embargo, esta es iterativa hasta␣
↪infinito, por lo que, sin no hay solución
    el algoritmo no converge.

    Parameters:
        problem (hanoi_states.ProblemHanoi): El problema de la Torre de Hanoi a␣
↪resolver.
        display (bool, optional): Muestra un mensaje de cuantos caminos se␣
↪expandieron y cuantos quedaron sin expandir.
                                    Por defecto es False.

    Returns:
        tree_hanoi.NodeHanoi: El nodo que contiene la solución encontrada.
    """

    depth = 1
    while True:
        # Ejecuto la busqueda para un límite dado
        last_node = depth_limited_search(problem, depth, display)
        if last_node != None:  # Si el último nodo no es nulo, entonces es la␣
↪solución.
            return last_node
        depth += 1
```

Algoritmos de búsqueda informada

[5]:
```
def heuristicFun(actualNode: hanoi_states.StatesHanoi, goal: hanoi_states.
↪StatesHanoi):
    """
    La función heurística es una implementación de la vista en clase:
        h(n) es un punto menos por cada disco ubicado en la posición correcta.
    """
    h_cost = 0

    # Al hacer zip(goal.rods, actualNode.rods) se obtiene un iterador, en donde␣
↪goalSubList y actualNodeSubLis son listas, de la forma [[5, 4], [3, 2], [1]]
```

```
    # Al hacer nuevamente zip, se obtiene los elementos de dichas listas.
    # Comparando los elementos, podemos agregar el costo para el caso que sean iguales.
    for goalSubList, actualNodeSubLis in zip(goal.rods, actualNode.rods):
        for goalSubListElement, actualNodeSubLisElement in zip(goalSubList, actualNodeSubLis):
            h_cost += -1 if goalSubListElement == actualNodeSubLisElement else 0
    return h_cost
```

Algoritmo búsqueda voraz primero el mejor

[6]:
```
def greedy_best_first_search(problem: hanoi_states.ProblemHanoi, display: bool = False):
    """
    Realiza una búsqueda voraz, utilizando una funcion heurística.

    Parameters:
        problem (hanoi_states.ProblemHanoi): El problema de la Torre de Hanoi a resolver.
        display (bool, optional): Muestra un mensaje de cuantos caminos se expandieron y cuantos quedaron sin expandir.
                                  Por defecto es False.

    Returns:
        tree_hanoi.NodeHanoi: El nodo que contiene la solución encontrada.
    """

    def heuristicFunWrapper(node: tree_hanoi.NodeHanoi, problem: hanoi_states.ProblemHanoi = problem):
        return heuristicFun(node.state, problem.goal)

    # Dado que los elementos los vamos insertar ordenados, creamos una lista FIFO con el nodo inicial.
    frontier = deque([tree_hanoi.NodeHanoi(problem.initial)])

    explored = set()
    while frontier:
        node = frontier.pop()

        explored.add(node.state)

        if problem.goal_test(node.state):
            if display:
                print(len(explored), "caminos se expandieron y", len(frontier), "caminos quedaron en la frontera")
            return node
```

```
        # Insertamos los elementos ordenados según la función heurística.
        frontier.extend([child for child in sorted(node.expand(problem),␣
↪key=heuristicFunWrapper, reverse=True)
                         if child.state not in explored and child not in␣
↪frontier])
    return None
```

Algoritmo búsqueda A*

[7]:
```
def a_star_search(problem: hanoi_states.ProblemHanoi, display: bool = False):
    """
    Realiza una búsqueda A*, utilizando una función heurística.

    Parameters:
        problem (hanoi_states.ProblemHanoi): El problema de la Torre de Hanoi a␣
↪resolver.
        display (bool, optional): Muestra un mensaje de cuantos caminos se␣
↪expandieron y cuantos quedaron sin expandir.
                                  Por defecto es False.

    Returns:
        tree_hanoi.NodeHanoi: El nodo que contiene la solución encontrada.
    """

    def totalCostPlusHeuristic(node: tree_hanoi.NodeHanoi, problem:␣
↪hanoi_states.ProblemHanoi = problem):
        return heuristicFun(node.state, problem.goal) + node.path_cost

    # Dado que los elementos los vamos insertar ordenados, creamos una lista␣
↪FIFO con el nodo inicial.
    frontier = deque([tree_hanoi.NodeHanoi(problem.initial)])

    explored = set()
    while frontier:
        node = frontier.pop()

        explored.add(node.state)

        if problem.goal_test(node.state):
            if display:
                print(len(explored), "caminos se expandieron y", len(frontier),␣
↪"caminos quedaron en la frontera")
            return node

        # Insertamos los elementos ordenados según la función heurística.
```

```
        frontier.extend([child for child in sorted(node.expand(problem),␣
↪key=totalCostPlusHeuristic, reverse=True)
                         if child.state not in explored and child not in␣
↪frontier])
    return None
```

1.1.6 Ejercicio 5:

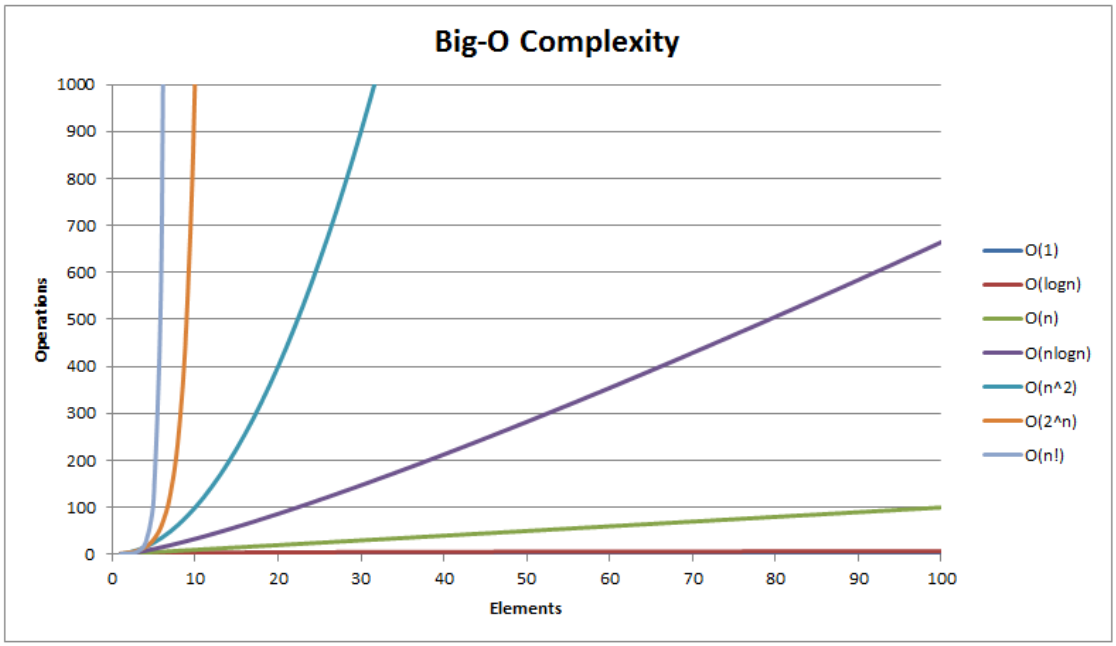
5. ¿Qué complejidad en tiempo y memoria tiene el algoritmo elegido?

La complejidad computacional es un concepto que se utiliza para medir la dificultad de resolver un problema de computación. En términos generales, la complejidad computacional se mide en términos de la cantidad de recursos necesarios para resolver un problema, como el *tiempo* o el *espacio*.

Existen varias métricas de complejidad computacional, entre las que se incluyen: - Tiempo de ejecución: Medida en términos de la cantidad de tiempo que se necesita para resolver un problema. - Espacio de almacenamiento: Medida en términos de la cantidad de espacio necesario para almacenar los datos y resultados durante la ejecución del programa. - Complejidad asintótica: Se utiliza para describir el comportamiento del tiempo de ejecución a medida que el tamaño del problema crece.

La complejidad computacional es importante en la teoría de la computación porque ayuda a entender qué problemas son resolubles y qué problemas son intractables. Un problema es considerado resoluble si existe un algoritmo eficiente para resolverlo. Un problema es considerado intractable si no existe un algoritmo eficiente para resolverlo, incluso si existe un algoritmo que lo resuelve.

La siguiente gráfica muestra la notación utilizada y su representación:



Para los algoritmos vistos, se tiene la siguiente información:

Algoritmo	Complejidad en tiempo	Complejidad en memoria	Descripción
BFS (breadth first search)	$O(b^d)$	$O(b^d)$	Este algorimo, en su peor caso, tiene que recorrer todos los nodos. Sea b el factor de ramificación (cantidad de hijos promedios, para este caso, nunca hay más de dos hijos) y d la profundidad del arbol, entonces tanto como 0para la complejdiad en tiempo, como en memoria, BSF tiene $O(b^d)$ en complejidad computacional. Sin embargo, es completo para un espacio finito.
DFS (depth first search)	$O(b^d)$	$O(bd)$	DFS mejora en complejidad de memoria (ya que no debe guardar todos los nodos en memoria de una rama) pero mantiene la complejidad en tiempo para el peor caso. No es óptimo, ya que puede no generar el camino más corto, a diferencia de BFS.

Algoritmo	Complejidad en tiempo	Complejidad en memoria	Descripción
IDDFS (iterative deepening depth first serach)	$O(b^d)$	$O(d)$	La busqueda iteractiva, en sí no tiene ninguna mejora para el por caso posible (lo que se suele medir en complejidad computacional), sin embargo, hay una gran mejora para el caso promedio, ya que se aprovecha de la poca memoria que utiliza. Para un factor de ramificación finito, es completo, pero no óptimo, igual a su ánálogo de forma no iterativa.
BeFS (best first search)	$O(b^d)$	$O(d)$	La búsqueda voraz, utilizando una heursística consistente no asegura un camino óptimo y no mejora los peores casos. Sin embargo, para el caso promedio sí, y de hecho es mucho mejor que las anteriroes, ya que puede llegar a ser de orden $O(logd)$ (recorre el árbol como en una búsqueda en profundidad pero orientada).
A* (a-star search)	$O(b^d)$	$O(b^d)$	Idem caso anterior pero muchas mejoras para los casos promedios. A demás, si la heurística es consistente, A* es completo y de óptima eficiencia.

1.1.7 Ejercicio 6:

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).

Busquedas

[8]:
```
import time # Funcionalidades para medir tiempos
import tracemalloc # Funcionalidad para medir memoria

max_disks = 5
num_iterations = 10 # Cantidad de iteraciones
algorithmsdf = pd.DataFrame(columns=["algorithm", "median elapsed_time (s)",␣
 ↪"median memory_peak (MB)", "median accumulated_cost (unit)"])
```

[9]:
```
def iteration(searchAlgoritm):
    """
    Ejecuta una iteración del algoritmo de búsqueda pasado como parámetro

    Args:
        searchAlgoritm (función): Algoritmo de búsqueda que devuelve el último␣
 ↪nodo del arbol.

    Returns:
        elapsed_time (float): Tiempo de ejecución
        memory_peak (float): Memoria utilizada
        accumulated_cost (int): Costo acumulado total del camino
    """

    initial_state = hanoi_states.StatesHanoi(
        [5, 4, 3, 2, 1], [], [], max_disks=max_disks)
    goal_state = hanoi_states.StatesHanoi([], [], [5, 4, 3, 2, 1],␣
 ↪max_disks=max_disks)

    # Crea una instancia del problema de la Torre de Hanoi
    problem_hanoi = hanoi_states.ProblemHanoi(
        initial=initial_state, goal=goal_state)

    # Para medir tiempo consumido
    start_time = time.perf_counter()
    # Para medir memoria consumida (usamos el pico de memoria)
    tracemalloc.start()

    # Resuelve el problema utilizando un algoritmo de búsqueda.
    last_node = searchAlgoritm(problem_hanoi)

    _, memory_peak = tracemalloc.get_traced_memory()
    memory_peak /= 1024*1024
```

```
    tracemalloc.stop()

    end_time = time.perf_counter()
    elapsed_time = end_time - start_time

    accumulated_cost = last_node.state.accumulated_cost if␣
 ↪isinstance(last_node, tree_hanoi.NodeHanoi) else 0

    return elapsed_time*10, round(memory_peak, 2), accumulated_cost
```

Algoritmo de busqueda en anchura (realizado en clase)

[10]:
```
results = []
for i in range(num_iterations):
    results.append(iteration(breadth_first_graph_search))

results = pd.DataFrame(results, columns=["elapsed_time (s)", "memory_peak␣
 ↪(MB)", "accumulated_cost (unit)"])
# Agrego los resultados al total
algorithmsdf.loc[len(algorithmsdf)] = ["breadth_first_graph_search"] + results.
 ↪median().values.tolist()
```

[11]:
```
print("Resultados BFS (breadth_first_graph_search)")
results.median()
```

```
Resultados BFS (breadth_first_graph_search)
```

[11]:
```
elapsed_time (s)            2.030525
memory_peak (MB)            0.230000
accumulated_cost (unit)    31.000000
dtype: float64
```

Algoritmo de busqueda en profundidad

[12]:
```
results = []
for i in range(num_iterations):
    results.append(iteration(depth_first_graph_search))

results = pd.DataFrame(results, columns=["elapsed_time (s)", "memory_peak␣
 ↪(MB)", "accumulated_cost (unit)"])
# Agrego los resultados al total
algorithmsdf.loc[len(algorithmsdf)] = ["depth_first_graph_search"] + results.
 ↪median().values.tolist()
```

[13]:
```
print("Resultados DFS (depth_first_graph_search)")
results.median()
```

```
Resultados DFS (depth_first_graph_search)
```

[13]:
```
elapsed_time (s)            0.56245
memory_peak (MB)            0.13000
accumulated_cost (unit)    81.00000
dtype: float64
```

Algoritmo de busqueda en profundidad iterativa

[14]:
```
results = []
for i in range(num_iterations):
    results.append(iteration(depth_iterative_limited_search))

results = pd.DataFrame(results, columns=["elapsed_time (s)", "memory_peak␣
 ↪(MB)", "accumulated_cost (unit)"])
# Agrego los resultados al total
algorithmsdf.loc[len(algorithmsdf)] = ["depth_iterative_limited_search"] +␣
 ↪results.median().values.tolist()
```

[15]:
```
print("Resultados IDDFS (depth_iterative_limited_search)")
results.median()
```

```
Resultados IDDFS (depth_iterative_limited_search)
```

[15]:
```
elapsed_time (s)           20.010351
memory_peak (MB)            0.150000
accumulated_cost (unit)    45.000000
dtype: float64
```

Algoritmo de busqueda voraz el primero siempre

[16]:
```
results = []
for i in range(num_iterations):
    results.append(iteration(greedy_best_first_search))

results = pd.DataFrame(results, columns=["elapsed_time (s)", "memory_peak␣
 ↪(MB)", "accumulated_cost (unit)"])
# Agrego los resultados al total
algorithmsdf.loc[len(algorithmsdf)] = ["greedy_best_first_search"] + results.
 ↪median().values.tolist()
```

[17]:
```
print("Resultados BeFS (greedy_best_first_search)")
results.median()
```

```
Resultados BeFS (greedy_best_first_search)
```

[17]:
```
elapsed_time (s)            0.555096
memory_peak (MB)            0.130000
accumulated_cost (unit)    81.000000
dtype: float64
```

Algoritmo de busqueda A*

[18]:
```
results = []
for i in range(num_iterations):
    results.append(iteration(a_star_search))

results = pd.DataFrame(results, columns=["elapsed_time (s)", "memory_peak␣
 ↪(MB)", "accumulated_cost (unit)"])
# Agrego los resultados al total
algorithmsdf.loc[len(algorithmsdf)] = ["a_star_search"] + results.median().
 ↪values.tolist()
```

[19]:
```
print("Resultados A* (a_star_search)")
results.median()
```

```
Resultados A* (a_star_search)
```

[19]:
```
elapsed_time (s)            0.516523
memory_peak (MB)            0.130000
accumulated_cost (unit)    81.000000
dtype: float64
```

Conclusion Como conclusión, se puede observar la siguiente tabla comparativa de los algoritmos:

[20]:
```
algorithmsdf.sort_values(by=["median elapsed_time (s)"], ascending=True)
```

[20]:

	algorithm	median elapsed_time (s)	median memory_peak (MB)	median accumulated_cost (unit)
4	a_star_search	0.516523	0.13	81.0
3	greedy_best_first_search	0.555096	0.13	81.0
1	depth_first_graph_search	0.562450	0.13	81.0
0	breadth_first_graph_search	2.030525	0.23	31.0
2	depth_iterative_limited_search	20.010351	0.15	45.0

1.1.8 Ejercicio 7:

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).

Tomando en cuenta, que el número de movimientos, dado que el costo es una unidad, es equivalente al costo acumulado. El costo ideal para este caso es:

[21]:
```
optimal_solution = (2**max_disks) - 1
print(f"Para {max_disks} discos, la solución óptima es {optimal_solution}.")
```

```
Para 5 discos, la solución óptima es 31.
```

La siguiente tabla muestra la comparativa entre los algoritmos (ordenada por el más óptimo empiricamente):

[22]:
```
algorithmsdf['distance to optimal'] = algorithmsdf['median accumulated_cost␣
↪(unit)'] - optimal_solution
algorithmsdf.iloc[:, [0,-1]].sort_values(by=['distance to optimal'],␣
↪ascending=True)
```

[22]:

	algorithm	distance to optimal
0	breadth_first_graph_search	0.0
2	depth_iterative_limited_search	14.0
1	depth_first_graph_search	50.0
3	greedy_best_first_search	50.0
4	a_star_search	50.0

Nota: En este punto podemos verificar que la heurística planteada para A no es consistente*