Introducción a la Inteligencia Artificial

TP1: Algoritmos de búsqueda en Torre de Hanoi

Profesor:
Dr. Ing. Facundo Adrián Lucianna

Ing. Pablo Martin Gomez Verdini
gomezpablo86@gmail.com

Ing. Diego Paciotti Iacchelli
diegopaciotti@gmail.com

Ing. Joaquín Gonzalez
joagonzalez@gmail.com

20 de septiembre de 2024

Índice

Índice de figuras

Índice de tablas

1. Instrucciones

En clase presentamos el problema de la torre de Hanoi (Figura 1). 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.

El entregable es, por un lado, un archivo de **txt/PDF/Word** con las respuestas y por otro, los archivos con el código implementado, tambien pueden enviar una Notebook con el contenido y la solución. Si además agregan los json para usar en el simulador, es mejor. Pueden subir el contenido o proporcionar un enlace a un repositorio público (**GitHub** o **GitLab**) con el contenido. No olvidar especificar en el entregable los autores del TP.

Para resolver este TP son libres de usar los recursos que crean necesarios. Pueden resolverlo en cualquier lenguaje de programación y de la forma que consideren apropiada.

Pueden ahorrar tiempo usando el código ya implementado en **Python** que se encuentra en el repositorio hanoi_tower. Si usan este código, solo deben implementar el algoritmo de búsqueda, pero es importante que lean el código y entiendan que es cada parte.

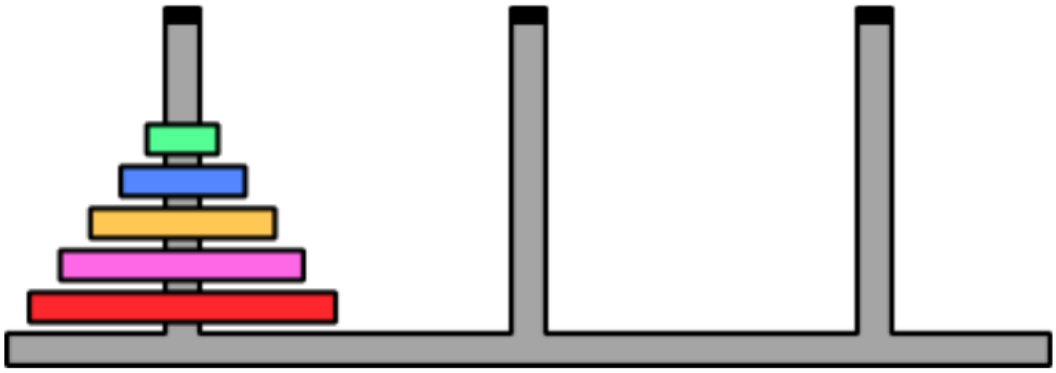

Figura 1: Torre de Hanoi de 5 discos en estado inicial.

2. Consignas

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

(2) ¿Cuáles son las propiedades del entorno de trabajo?

(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.

(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.

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

(6) Si la solución óptima es 2k -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).

3. Desarrollo

3.1. Consigna 1

Performance
El rendimiento se mide en función de la eficiencia con la que se resuelve el problema. Específicamente, podemos evaluar:

- Número mínimo de movimientos: la solución óptima requiere $2^n - 1$ movimientos para n discos.
- Tiempo de resolución: el tiempo que tarda el agente en encontrar la solución.
- Cantidad de memoria utilizada por el programa.
- Cantidad de caminos explorados.
- Cantidad de fronteras sin expandir.

Environment
El entorno esta compuesto por los discos de distintos diámetros y varillas con la disposición acorde a las reglas del juego.

Actuators
Los actuadores permiten que el agente realice las siguientes acciones: mover los discos para ejecutar un cambio de estado a uno permitido por el juego.

Sensors
Por sensores podemos considerar la detección de la posición de los discos en las varillas para mapearlos al estado actual y conocer las restricciones de movimiento.

3.2. Consigna 2

Podemos describir al entorno de trabajo como:

- **Totalmente observable**: El agente puede ver toda la disposición de los discos en los tres postes en cualquier momento.
- **Determinístico**: Cada acción tiene un resultado predecible (mover un disco siempre tiene el mismo efecto).
- **Secuencial**: La secuencia de acciones importa, ya que un movimiento afecta el estado siguiente.
- **Estático**: El entorno no cambia mientras el agente está deliberando o ejecutando acciones.
- **Discreto**: Hay un número finito y definido de estados (la cantidad de discos y posiciones posibles es limitada).
- **No competitivo**: No hay otros agentes involucrados en el entorno que influyan en el resultado.

3.3. Consigna 3

- **Estado**: Un estado está representado por la configuración actual de los discos en los tres postes. Ejemplo: $[[3, 2, 1], [], []]$ representa tres discos en el primer poste.
- **Espacio de estados**: El espacio de estados es el conjunto de todas las posibles configuraciones de discos en los tres postes. Para n discos, hay 3^n posibles estados.
- **Árbol de búsqueda**: El árbol de búsqueda representa la exploración del espacio de estados. Cada nodo del árbol es un estado y las aristas son las acciones (mover un disco de un poste a otro). El árbol comienza con el estado inicial y se expande al aplicar acciones, generando nuevos estados.
- **Nodo de búsqueda**: Un nodo de búsqueda es una estructura que contiene:
  - El estado actual del entorno.
  - Un registro de la acción que llevó a este estado desde el nodo padre.
  - La secuencia de acciones para llegar a este estado desde el inicio.
- **Objetivo**: El estado objetivo es tener todos los discos en el poste de destino (por ejemplo, $[[], [], [3, 2, 1]]$).
- **Acción**: Una acción es mover un disco de un poste a otro, respetando las reglas del problema (Solo se puede mover un disco cada vez y un disco más grande no puede ir encima de uno más pequeño).
- **Frontera**: La frontera es el conjunto de nodos que aún no han sido explorados en el árbol de búsqueda. Estos son los nodos en los que el agente tiene que decidir cuál expandir a continuación para explorar nuevos estados.

3.4. Consigna 4

Se implementaron, además de la búsqueda por anchura, los algoritmos de **A*** y **Greedy** con 3 heurísticas distintas. Las combinatorias analizadas en este trabajo se detallan en la Tabla 1.

En los bloques de código 2 3 4 se detallan las implementaciones de cada uno junto con las funciones heurísticas.

Algoritmo	**Heurística**
breadth-first-graph-search	Sin heurística
astar-search	heuristic-func-astar_1 + costo
astar-search	heuristic-func-astar_2 + costo
astar-search	heuristic-func-greedy + costo
greedy-search	heuristic-func-astar_1
greedy-search	heuristic-func-astar_2
greedy-search	heuristic-func-greedy

Tabla 1: Combinación de algoritmos y heurísticas implementadas que serán comparadas.

```python
def heuristic_func_astar_1 (nodeState: hanoi_states.StatesHanoi) -> int:
    return (-1 * sum(nodeState.rods[-1]))

def heuristic_func_astar_2(nodeState: hanoi_states.StatesHanoi, num_of_disks: int = 5) ->
↪  int:
    def compute_rod(current_rod, number_of_disks):
        goal_rod = list(range(number_of_disks, 0, -1))
        result = [0] * number_of_disks
        for i in range(min(len(current_rod), number_of_disks)):
            if current_rod[i] == goal_rod[i]:
                result[i] = 1
        return result
    num_of_disks = max(max(rod) for rod in nodeState.rods if len(rod) > 0)
    current_target_rod_disks = nodeState.rods[-1]
    computed_rod = compute_rod(current_target_rod_disks, num_of_disks)
    reward = sum([-2**(num_of_disks-idx-1)*i for idx, i in enumerate(computed_rod)])
    return reward

def heuristic_func_greedy (nodeState: hanoi_states.StatesHanoi) -> int:
    return sum(nodeState.rods[-1])-sum(nodeState.rods[0])
```

Figura 2: Implementación de distintas heurísticas para algoritmos A* y Greedy.

- La primera heurística evaluada simplemente busca dar mas peso a los discos ubicados en la ultima varilla considerando el diámetro de los mismos.
- En la segunda, además de buscar la mayor cantidad de discos en la última varilla, busca que el orden sea el más adecuado. Pondera mejor cuando el disco de mayor diámetro ya está insertado en la misma y con el apilamiento de los discos de forma ordenada, de esta forma garantiza estar mas cerca de la solución, ya que son menos los estados para llegar a la solución.
- La tercer heurística funciona en ponderar estados en donde la mayor cantidad de discos esta en la última varilla y donde la cantidad de discos en la primera es menor, esto con el objetivo de alejarnos lo mas rápido posible del estado inicial y acercarnos al final. La mayor diferencia con las dos anteriores es que el mejor valor es el mas alto y no el mas bajo. Esta lógica se usa dando vuelta el orden de la PQ, para ir retirando los estados de la cola en ese orden.

Las implementaciones completas que se detallan a continuación pueden encontrarse en el repositorio de GitHub IIA-TP1.

El algoritmo `greedy_search` implementa una búsqueda voraz que prioriza los nodos según una función heurística, sin considerar el costo acumulado del camino. A continuación se presenta la implementación del algoritmo:

```
def greedy_search(problem: hanoi_states.ProblemHanoi, heuristic_func: Callable, display:
↪  bool = False):
    def f(new_node):
        return heuristic_func(new_node.state)
    node = tree_hanoi.NodeHanoi(problem.initial)
    if problem.goal_test(node.state):
        return node
    if "greedy" in getattr(heuristic_func, '__name__', 'Unknown'):
        frontier = aima.PriorityQueue(order='max', f=f)
    else:
        frontier = aima.PriorityQueue(order='min', f=f)
    frontier.append(node)
    reached = {node.state: node}
    while len(frontier) > 0:
        node = frontier.pop()
        if problem.goal_test(node.state):
            if display:
                print(len(reached), "caminos se expandieron y", len(frontier), "caminos
                ↪  quedaron en la frontera")
            return (node, len(reached), len(frontier))
        for child in node.expand(problem):
            s = child.state
            if s not in reached or f(child) < f(reached[s]):
                reached[s] = child
                if child in frontier:
                    del frontier[child]
                frontier.append(child)
    return "failure"
```

Figura 3: Implementación de algoritmo Greedy.

El algoritmo `astar_search` implementa la búsqueda A*, que combina la función heurística con el costo acumulado del camino (`path_cost`) para priorizar los nodos. A continuación se presenta la implementación del algoritmo:

```
def astar_search(problem: hanoi_states.ProblemHanoi, heuristic_func: Callable, display:
↪  bool = False):
    def f(new_node):
        return heuristic_func(new_node.state) + new_node.path_cost
    node = tree_hanoi.NodeHanoi(problem.initial)
    if problem.goal_test(node.state):
        return node
    if "greedy" in getattr(heuristic_func, '__name__', 'Unknown'):
        frontier = aima.PriorityQueue(order='max', f=f)
    else:
        frontier = aima.PriorityQueue(order='min', f=f)
    frontier.append(node)
    reached = {node.state: node}
    while len(frontier) > 0:
        node = frontier.pop()
        if problem.goal_test(node.state):
            if display:
                print(len(reached), "caminos se expandieron y", len(frontier), "caminos
                ↪  quedaron en la frontera")
            return (node, len(reached), len(frontier))
        for child in node.expand(problem):
            s = child.state
            if s not in reached or f(child) < f(reached[s]):
                reached[s] = child  # Update the reached dictionary with the better node
                if child in frontier:
                    del frontier[child]
                frontier.append(child)
    return "failure"
```

Figura 4: Implementación de algoritmo A*.

3.5. Consigna 5

En la Figura 5 se detalla el tiempo de ejecución promedio para diez iteraciones de cada tupla (algoritmo, heurística) agrupadas por cantidad de discos (Tabla 1). Se han evaluado diferentes escenarios para el problema de la torre de Hanoi utilizando desde 3 hasta 7 discos inclusive.

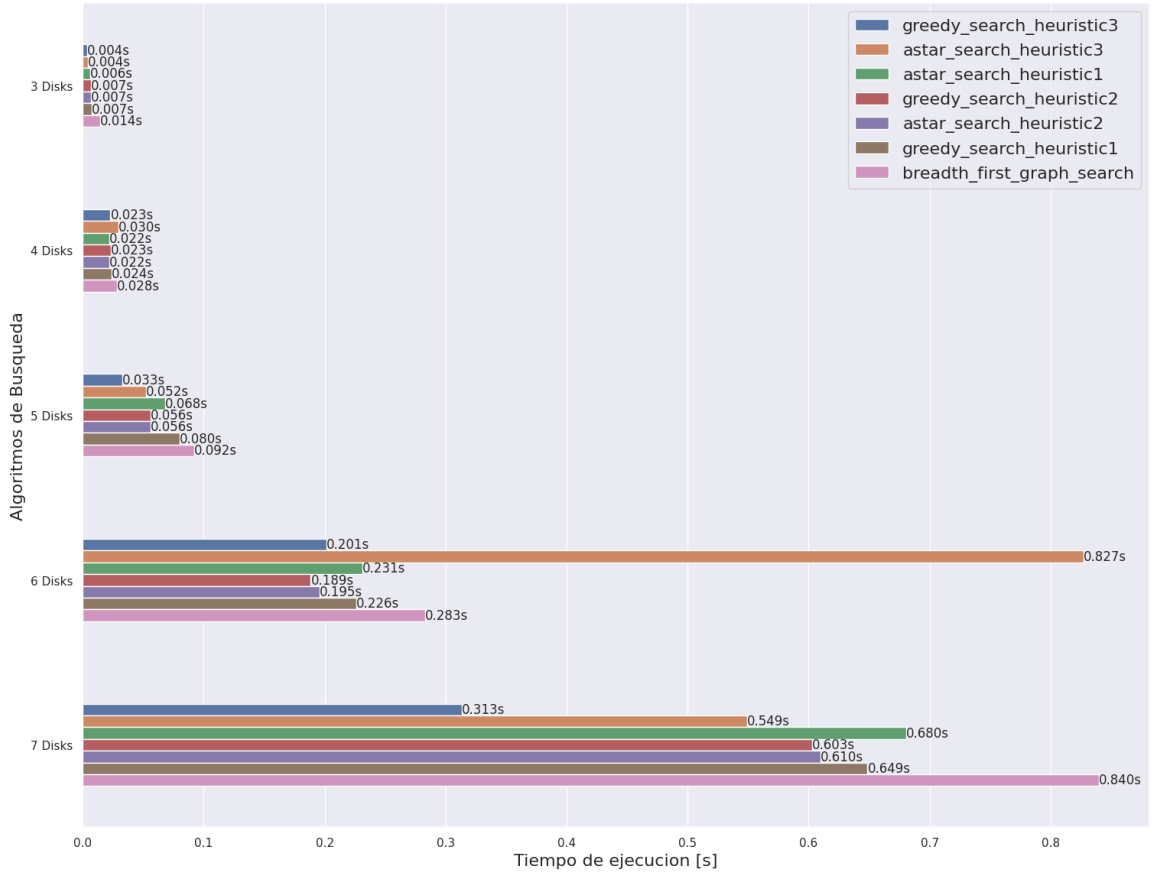

Figura 5: Tiempo de ejecución promedio de algoritmo de búsqueda por cantidad de discos.

La siguiente tabla expande lo representado en el punto anterior añadiendo la media, el desvío estándar y la varianza a los valores obtenidos durante las distintas ejecuciones de código.

model-name	disks	mean	std	var
greedy-search-heuristic3	3	0.00380225229891948	0.000639552477492111	4.09027371466298E-07
astar-search-heuristic3	3	0.00436415679869242	0.00083515412263792	6.97482408559113E-07
astar-search-heuristic1	3	0.00591409879853018	0.00048074694182065	2.31117622069907E-07
greedy-search-heuristic2	3	0.00668829929782078	0.000538325974821556	2.89794855167578E-07
astar-search-heuristic2	3	0.00691464400442783	0.000599897791651011	3.5987736042776E-07
greedy-search-heuristic1	3	0.00745603929681238	0.000595875560525966	3.55067683632134E-07
breadth-first-graph-search	3	0.0144131398963509	0.0129995698056152	0.000168988815131062
astar-search-heuristic1	4	0.0221047011960763	0.00173316856506541	3.0038732749309E-06
astar-search-heuristic2	4	0.0221325911028544	0.00226585417535291	5.13409514396423E-06
greedy-search-heuristic3	4	0.0228600806934992	0.00138151454295221	1.90858243238846E-06
greedy-search-heuristic2	4	0.0230835413036402	0.00624368089148542	3.89835510747002E-05
greedy-search-heuristic1	4	0.0239008230040781	0.00171921569225034	2.95570259647981E-06
breadth-first-graph-search	4	0.0283210878027603	0.00209483964114462	4.38835312211093E-06
astar-search-heuristic3	4	0.0297726086981129	0.00205330751314024	4.21607174351815E-06
greedy-search-heuristic3	5	0.0328385533997789	0.00127577465540589	1.62760097137602E-06
astar-search-heuristic3	5	0.0520934203959769	0.00229767673733676	5.2793183892985E-06
greedy-search-heuristic2	5	0.0561505252000643	0.00222339798451114	4.94349859752819E-06
astar-search-heuristic2	5	0.0562292495014844	0.00310000506776623	9.6100314201763E-06
astar-search-heuristic1	5	0.0679700622975361	0.00259039809368364	6.71016228375985E-06
greedy-search-heuristic1	5	0.0799150512029883	0.0151642252348644	0.000229953726973699
breadth-first-graph-search	5	0.091996741303592	0.00117209977249552	1.37381787668404E-06
greedy-search-heuristic2	6	0.188516382206581	0.00293225730080604	8.59813287813031E-06
astar-search-heuristic2	6	0.195481654294417	0.0082960735709125	6.88248366939929E-05
greedy-search-heuristic3	6	0.201347590595833	0.00299777362960113	8.98664673433196E-06
greedy-search-heuristic1	6	0.225949639902683	0.0167774324508293	0.000281482239642139
astar-search-heuristic1	6	0.231269434292335	0.0196990802415738	0.000388053762363962
breadth-first-graph-search	6	0.282778792607132	0.0114806240924693	0.000131804729552586
astar-search-heuristic3	6	0.827043122594478	0.019216070442869	0.000369257363265303
greedy-search-heuristic3	7	0.313438907099771	0.0133769384599021	0.000178942482560007
astar-search-heuristic3	7	0.548926913796458	0.0317540148141832	0.00100831745681937
greedy-search-heuristic2	7	0.602556956600165	0.0484313306667713	0.00234559379015414
astar-search-heuristic2	7	0.609884384603356	0.0258089336317823	0.000666101055209745
greedy-search-heuristic1	7	0.648626189594506	0.036978508069272	0.00136741005902921
astar-search-heuristic1	7	0.680330324303941	0.0481628223276583	0.00231965745456558
breadth-first-graph-search	7	0.839797900407575	0.0584475417634813	0.00341611513819389

Tabla 2: Estadísticos del tiempo de ejecución.

En la Figura 6 se detalla el consumo de memoria promedio para diez ejecuciones de cada tupla (algoritmo, heurística) agrupadas por cantidad de discos (Tabla 1).

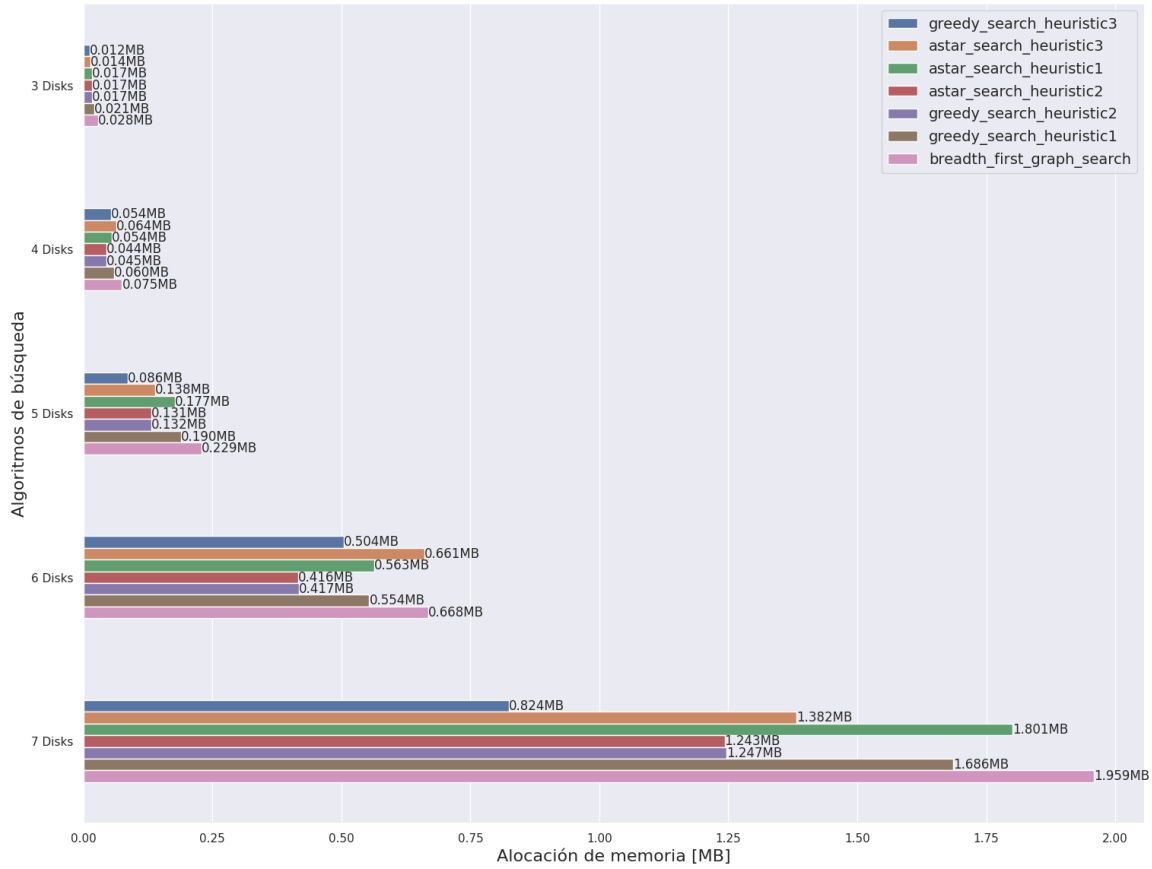

Figura 6: Uso de memoria promedio de algoritmo de búsqueda por cantidad de discos.

La siguiente tabla expande lo representado en el punto anterior añadiendo la media, el desvío estándar y la varianza a los valores obtenidos durante las distintas ejecuciones de código.

model-name	disks	mean	std	var
greedy-search-heuristic3	3	0.011952972412109375	0.0004525602381934392	2.0481076919370244e-07
astar-search-heuristic3	3	0.01387939453125	0.0005266652825079295	2.773763197991572e-07
astar-search-heuristic1	3	0.01665802001953125	0.0004187576252792153	6.7036404895286e-07
greedy-search-heuristic2	3	0.017046356201171876	0.0004742611083411576	5.494680307391612e-07
astar-search-heuristic2	3	0.017057037353515624	0.000749279656590286	5.61420003780057e-07
greedy-search-heuristic1	3	0.020647239685058594	0.0010229741363808598	1.046476083704166e-06
breadth-first-graph-search	3	0.02835502624511719	0.0026223645803858343	6.876795992462173e-06
astar-search-heuristic2	4	0.04447364807128906	0.0	0.0
greedy-search-heuristic2	4	0.04452705383300781	0.0	0.0
greedy-search-heuristic3	4	0.053971290588378906	0.0	0.0
astar-search-heuristic1	4	0.05423736572265625	0.0	0.0
greedy-search-heuristic1	4	0.05985450744628906	0.0	0.0
astar-search-heuristic3	4	0.06413078308105469	3.6189395830173735e-05	1.309672370552996e-09
breadth-first-graph-search	4	0.074810791015625	0.0006417586193884189	4.118541255593295e-07
greedy-search-heuristic3	5	0.08616790771484376	0.002972355710851626	8.834898471832277e-06
astar-search-heuristic2	5	0.13115005493164061	0.0017151667581804244	3.067986108362682e-06
greedy-search-heuristic2	5	0.13166122436523436	0.0044175189176699076	1.9514473387971512e-05
astar-search-heuristic3	5	0.13824234008789063	1.68883847207469e-05	2.8521753847595733e-10
astar-search-heuristic1	5	0.1773275375366211	0.0064127609411068365	4.112350288778544e-05
greedy-search-heuristic1	5	0.18956336975097657	0.005988138696699462	3.585780505090953e-05
breadth-first-graph-search	5	0.22885627746582032	0.009243994755996346	8.545143904888794e-05
astar-search-heuristic2	6	0.41559791564941406	0.007256786662049256	5.266095265849599e-05
greedy-search-heuristic2	6	0.4170047760009766	0.014521598233454494	0.00021087681525386865
greedy-search-heuristic3	6	0.5043487548828125	0.007527394332676196	5.666166543960571e-05
greedy-search-heuristic1	6	0.5539630889892578	0.010055826788011018	0.00010111965239047997
astar-search-heuristic1	6	0.5634061813354492	0.012507055198908145	0.00015642642974853524
astar-search-heuristic3	6	0.6610918045043945	0.024432177776731963	0.0005969313109138351
breadth-first-graph-search	6	0.6679931640625	0.01793738208448554	0.00032174967604482274
greedy-search-heuristic3	7	0.8240018844604492	0.01957122526495811	0.00038303285837173453
astar-search-heuristic2	7	1.2428443908691407	0.018949521886066596	0.0003590843797105169
greedy-search-heuristic2	7	1.246760940551758	0.03301940622815868	0.0010902811876601641
astar-search-heuristic3	7	1.3818492889404297	0.02893991161582332	0.0008375184843316657
greedy-search-heuristic1	7	1.6862363815307617	0.031834605198609756	0.0010134420881513515
astar-search-heuristic1	7	1.8005821228027343	0.02581098790170241	0.0006662070964618281
breadth-first-graph-search	7	1.9590326309204102	0.040128673197045925	0.0016103104125553122

Tabla 3: Estadísticos de memoria utilizada.

La función heurística **heuristic-func-greedy** (2) se pensó buscando maximizar el valor de $h(nodo)$ para los estados más próximos a la solución deseada, en vez de buscar minimizar el valor de $h(nodo)$ como se hizo en las otras dos funciones heurísticas. Para esto, se cambió el orden de las prioridades en la cola PQ. Al ejecutarse el algoritmo **A*** (4), $f(nodo)$ resulta de la suma entre $h(nodo)$ y el costo del camino. $f(nodo) = costo(nodo) \downarrow + h(nodo) \uparrow$
Estas funciones valoran de manera inversa a cada nodo. Por un lado, $h(n)$ otorgará un valor mayor cuanto más cerca del estado óptimo esté el nodo, y por el otro $f(n)$ devolverá un número menor cuanto más cerca del resultado final esté el nodo bajo análisis. Esto genera una incompatibilidad entre ambas métricas sumado a que la PQ prioriza los valores mas altos. *frontier = aima.PriorityQueue(order='max', f=f)*

3.6. Consigna 6

Distancia al resultado óptimo

A continuación, se presenta un gráfico[7] que permite visualizar la distancia, en cantidad de estados, entre el camino obtenido por cada algoritmo de búsqueda y el camino óptimo que surge de la expresión $2^k - 1$. Además, en las Figuras [8] [9] y [10] se muestran, utilizando el mismo corte por disco, los costos del camino, fronteras no exploradas y nodos explorados respectivamente.

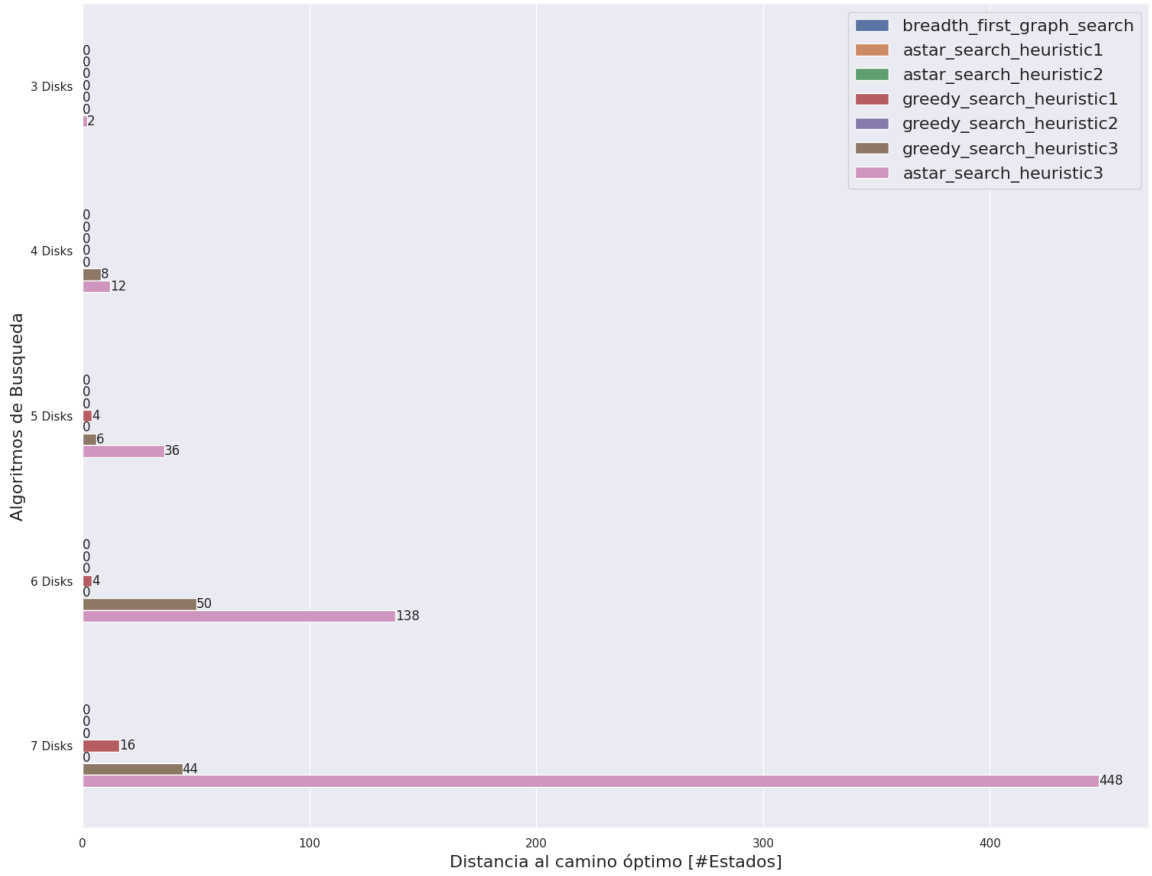

Figura 7: Distancia al camino óptimo para cada algoritmo de búsqueda por cantidad de discos.

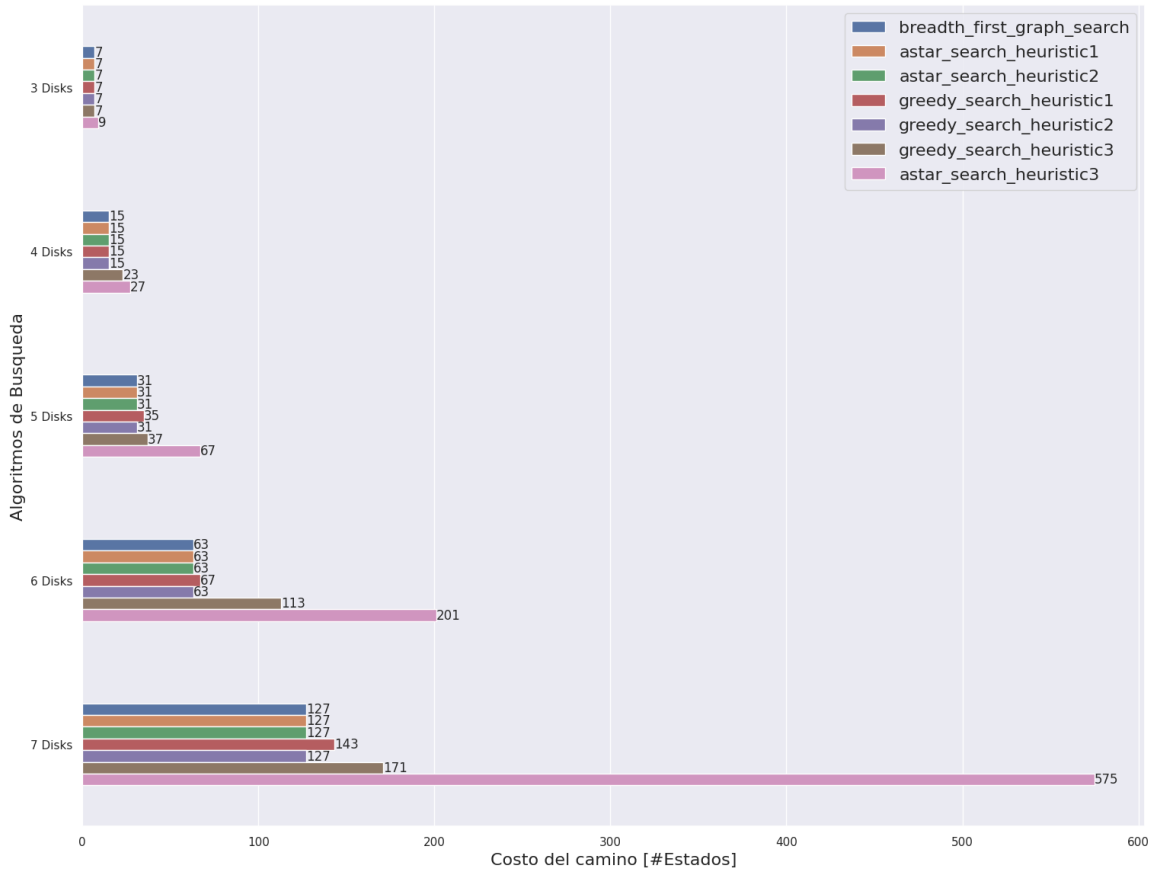

Figura 8: Costo del camino para cada algoritmo de búsqueda por cantidad de discos.

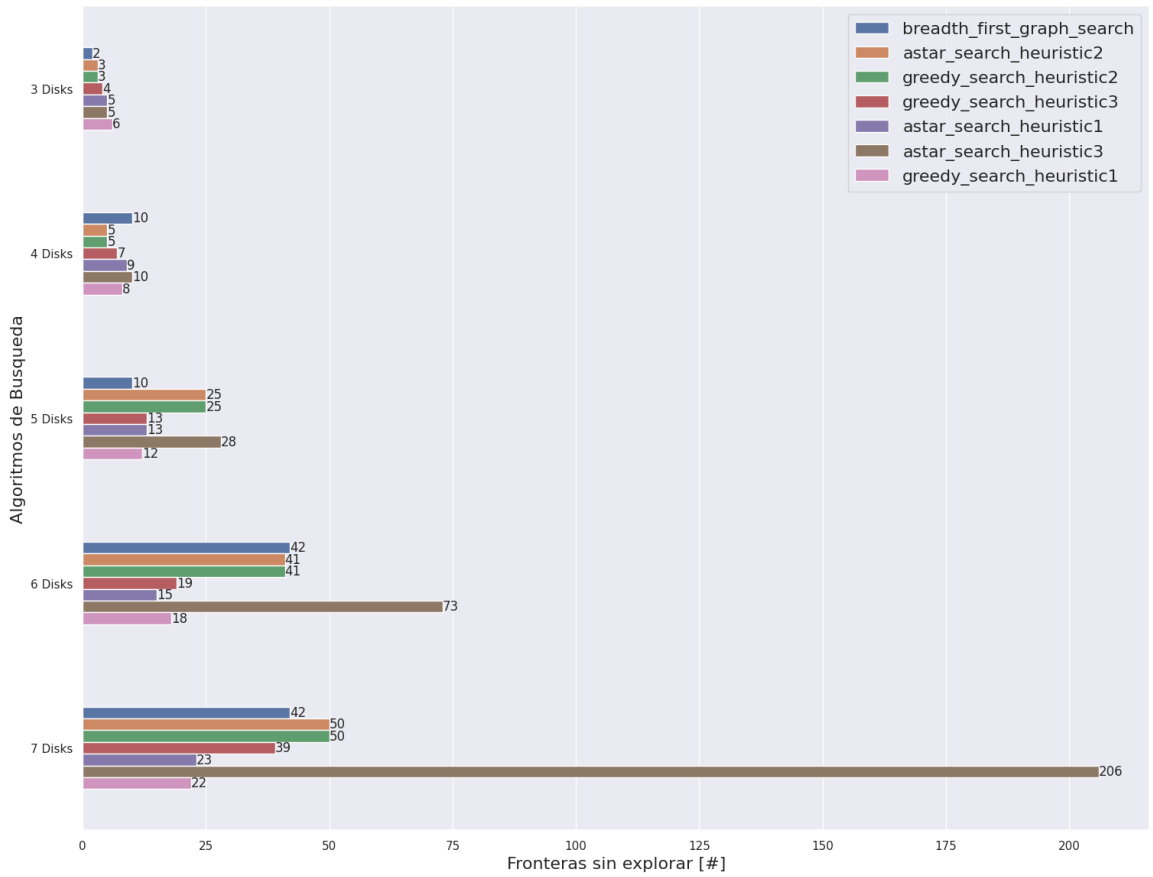

Figura 9: Fronteras sin explorar para cada algoritmo de búsqueda por cantidad de discos.

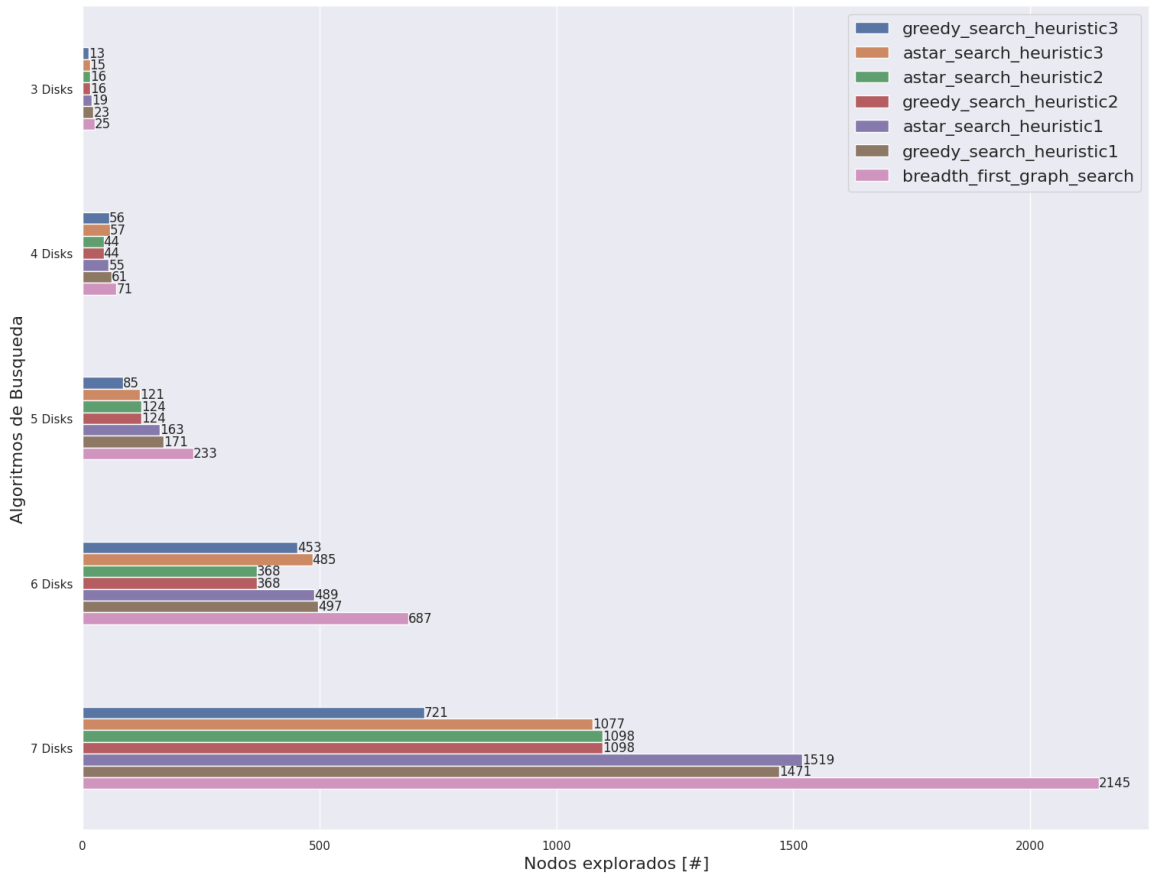

Figura 10: Nodos explorados para cada algoritmo de búsqueda por cantidad de discos.

3.7. Conclusiones

Como se observa en las Figuras [5] y [6], el algoritmo más rápido para encontrar la solución óptima y más eficiente en utilización de memoria es **A***. Es importante aclarar que, si bien **Greedy** con la función $h(nodo)$ número 3 es ligeramente más eficiente en las variables antes mencionadas, la solución encontrada no es la óptima y acarrea un comportamiento inconsistente como se menciona en la sección 3.5 y como se puede observar en las distancias a los caminos óptimos de las soluciones que genera [7], por lo que no se la considera para elaborar el ranking.

Como se puede observar en la Figura 7 las distancias no nulas a la solución óptima se dan con las variantes **Greedy** y con **A*** con la heurística 3. El rendimiento anteriormente detallado es consistente cuando se observan los costos en la Figura 8, donde para distintos discos **A*** y **breadth_first_graph_search** son los que dan mejores valores.

Lo mismo pasa con las figuras Figura 9 y Figura 10 donde **A*** y **Greedy** son los que exploran menor cantidad de nodos, siendo consistente con los tiempos bajos en comparación al resto.

Para el caso de la heurística podemos inferir que mientras mas optimizada este para el caso de uso (entre **A*** o **Greedy**) mejor va a responder el algoritmo.

A. Ejecución y uso de código

Para facilitar el trabajo colaborativo, se implementó una base de datos y se modificó el script base para poder guardar los resultados de las corridas allí. Luego, el análisis de los datos se realizo utilizando pandas sobre la base de datos SQL. Este esquema se puede ver en la Figura 11.

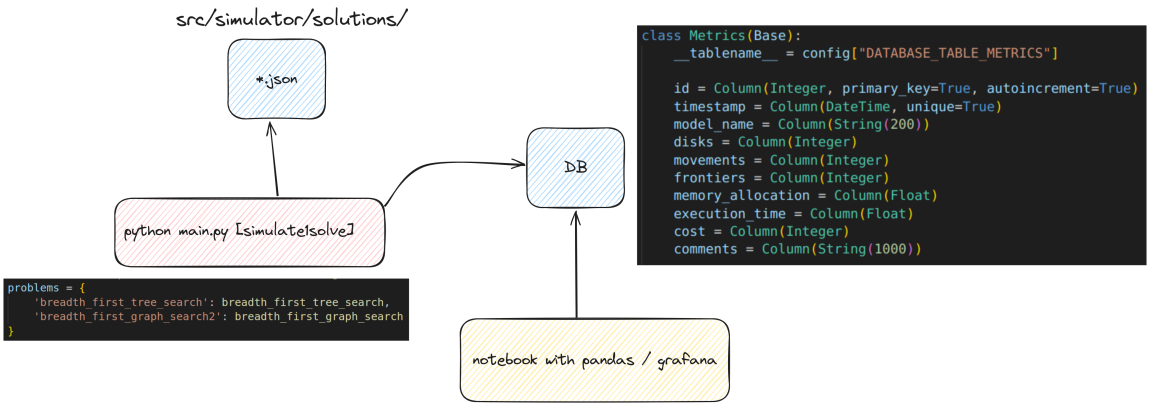

Figura 11: Esquema de trabajo.

Para el correcto uso de los scripts en `Python` adjunto, se deben realizar los siguientes pasos:

```
# clonar github repository
git clone git@github.com:FIUBA-CEIA-18Co2024/IIA-TP1.git

# crear y activar virtual environment
python -m venv venv
venv\Scripts\activate

# instalar dependencias
pip install -r requirements.txt

# en el archivo main seleccionar algoritmos y heuristicas a utilizar
problems = {
            'breadth_first_graph_search': breadth_first_graph_search,
            'astar_search_heuristic1': astar_search_heuristic1,
            'astar_search_heuristic2': astar_search_heuristic2,
            'greedy_search_heuristic1': greedy_search_heuristic1,
            'greedy_search_heuristic2': greedy_search_heuristic2,
        }

# Resolver hanoi con parametros definidos en main.py para todos los algoritmos definidos en
↪  diccionario problems
python main.py solve [opcional|numero_de_discos]

# Lo mismo que opcion anterior pero guarda los resultados en una base de datos externa para
↪  posterior analisis
python main.py solve-db [opcional|numero_de_discos]

# Ejecutar desde 3 discos hasta [numero_de_discos] 10 veces todos los algoritmos por cada
↪  variante para poder analizar datos posteriormente
python main.py solve-db-m [numero_de_discos]

Output:
--------------------------------------------------
Solving problem using breadth_first_tree_search
233 caminos se expandieron y 10 caminos quedaron en la frontera
Tiempo que demoró execute_algorithm: 0.052971 [s]
Maxima memoria ocupada: 0.32 [MB]
Longitud del camino de la solución: 31.0
--------------------------------------------------
Solving problem using breadth_first_graph_search2
233 caminos se expandieron y 10 caminos quedaron en la frontera
Tiempo que demoró execute_algorithm: 0.067039 [s]
Maxima memoria ocupada: 0.29 [MB]
Longitud del camino de la solución: 31.0

# Simulaciones
# Correr simulacion. Se generaran dos archivos JSON por algoritmo de busqueda utilizado en
↪  ./src/simulator/solutions llamados initial_state_{name}.json y sequence_{name}.json.
↪  Entonces, para ejecutar una simulación realizamos lo siguiente:

# correr simulacion con ultima solucion generada
python main.py simulate breadth_first_tree_search
python main.py simulate breadth_first_graph_search2
```


Figura 12: Instrucciones para instalar dependencias y ejecución del código.