Programa 2: Las Torres de Hanói

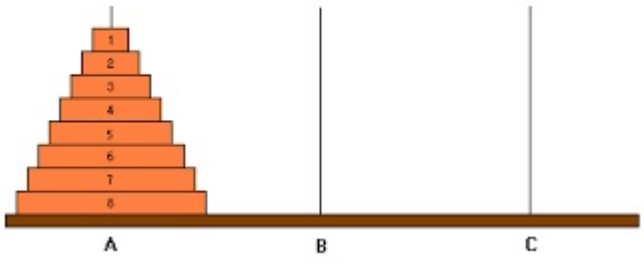
Gil Ramírez Bruno
Moguel Silva José Miguel
Ramirez Santamaria Isaí
Zacarias Daniel Luis Alberto

30 de Marzo de 2023

Índice

1. Planteamiento del problema

El juego, en su forma más tradicional, consiste en tres postes verticales. En uno de los postes se apila un número indeterminado de discos perforados por su centro. Los discos se apilan en tamaño decreciente de abajo hacia arriba. No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio (desde la base del poste hacia arriba) en uno de los postes, quedando los otros dos postes vacíos.



El juego consiste en pasar todos los discos desde el poste ocupado a uno de los otros postes vacíos. Para realizar este objetivo, es necesario seguir tres simples reglas:

1. Solo se puede mover un disco a la vez y para mover otro los demás tienen que estar en postes (es decir, no pueden estar afuera de las torres los demás discos).

2. Un disco de mayor tamaño no puede estar sobre uno más pequeño que él mismo.

3. Solo se puede desplazar el disco que se encuentre arriba en cada poste.

2. Metodología

2.1. Estados inicial y final

La torre de Hanói se representará en nuestro algoritmo como un arreglo de tres arreglos dentro, donde cada sub arreglo representa un poste y sus contenidos representan los discos; dichos son representados con números de manera decreciente, siendo estos su tamaño respectivamente.

Dicho lo anterior, el estado inicial está representado con el primer sub arreglo lleno con los números de sus respectivos discos. Por ejemplo: $[[4, 3, 2, 1], [], []]$ En cuanto el estado final, está representado con el último sub arreglo lleno con los números de los discos. Por ejemplo: $[[], [], [4, 3, 2, 1]]$

2.2. Función objetivo

La función objetivo solo va a checar que el estado actual sea igual al estado final, siendo que los números de los discos estén en el último sub arreglo en el orden correcto, tal y como se requiere para completar el juego.

2.3. Función sucesor

La función sucesor se encarga de generar los hijos del estado en el que se encuentra el juego. Dentro de esta función se buscan los posibles movimientos que se pueden realizar de cada poste, asegurándose que el disco que está hasta arriba no se apile en su mismo poste y cumpliendo que no se coloque un disco sobre otro más pequeño que el que se va a mover. Si el movimiento del disco cumple con estas dos condiciones, se añade como hijo sucesor y la función continúa buscando otros posibles caminos para tomarlos en cuenta posteriormente.

2.4. Función heurística

La función heurística que se basa en la búsqueda A*, se encarga de realizar un cálculo de costo estimado para ordenar la lista de estados sucesores de acuerdo a este algoritmo y decidir por cuál movimiento optar primero, además, también continúa haciendo cálculos de otros estados sucesores para optar por ellos posteriormente.
El costo estimado se realiza checando si los discos no están en el poste objetivo o si el disco está en el poste objetivo, pero no está en la posición correcta, sumando una unidad al costo estimado si alguna de estas condiciones es afirmativa.

3. Diseño y descripción del algoritmo

La meta del algoritmo es que además de completar el juego moviendo cada uno de los discos del primer poste al tercero, se encuentre la forma de completarlo con el menor número de pasos posible, ya que dicho algoritmo consiste en ser óptimo.
Para ello se empezó por representar los postes por medio de un arreglo de tres arreglos, donde cada sub arreglo representa un poste y sus contenidos representan los discos; dichos son representados con números de manera decreciente, siendo estos su tamaño respectivamente. Por ejemplo, el arreglo siguiente de una torre de Hanói de 4 discos nos dice que el disco más pequeño está en el segundo poste, el que le sigue está en el tercer poste, y los restantes están en el primer poste: $[[4, 3], [1], [2]]$

Para la resolución de este juego, se tomó en cuenta el estado inicial que son los discos en la primera torre, un costo inicial empezando desde 0 para proceder a la búsqueda heurística, y un espacio para obtener el padre de los hijos sucesores que se vayan adquiriendo al momento de encontrar la solución y así obtener la trayectoria. También se considera un arreglo de estados ya visitados para no acabar en un ciclo infinito que nunca encuentre la solución.

Luego se procedió con un ciclo que continúa hasta que la frontera de la búsqueda quede vacía o hasta que la función objetivo retorne "True" de que encontró la solución al juego. En dicha función se hace la comparación entre el estado actual del juego que se obtuvo de la frontera; con el estado final que es el objetivo. Si la función retorna "True", cumple con la condición y se almacena el camino que formaron los padres del estado solución. Este

camino es retornado para ser interpretado o bien impreso en la terminal o representado en una interfaz gráfica animada que muestra paso a paso los movimientos realizados hasta la solución del juego.

Después se continuó con la función sucesor, el cual se encarga de generar los estados hijos que cumplen con la condición de no volver a colocar un disco en el mismo poste donde estaba antes de su movimiento y con no colocar un disco sobre otro más pequeño que este, siendo estos estados totalmente válidos para avanzar en el juego.

Finalmente, para llevar a cabo la búsqueda heurística A*, al obtener los estados sucesores gracias a la función sucesor, se acumula un costo que se inició desde 0 y aumenta una unidad por cada movimiento realizado en cada uno de los estados sucesores. Este costo es el denominado camino recorrido o bien costo acumulado. Dicho será utilizado para ordenar la lista de sucesores que está dentro de la frontera basándose en la función heurística, la cual se encarga de realizar un cálculo estimado de cuantos posibles pasos faltan a cada sucesor para llegar al estado objetivo, tomando en cuenta si los discos no están en el poste objetivo o si el disco está en el poste objetivo, pero no está en la posición correcta; se suma una unidad al costo estimado si alguna de estas condiciones es afirmativa.
Una vez obtenido el costo estimado, se suma con el costo acumulado y estos dos valores se utilizan de criterio para ordenar la lista de sucesores, y así, determinar a cuál de todos darle prioridad primero para obtener el camino más óptimo posible.

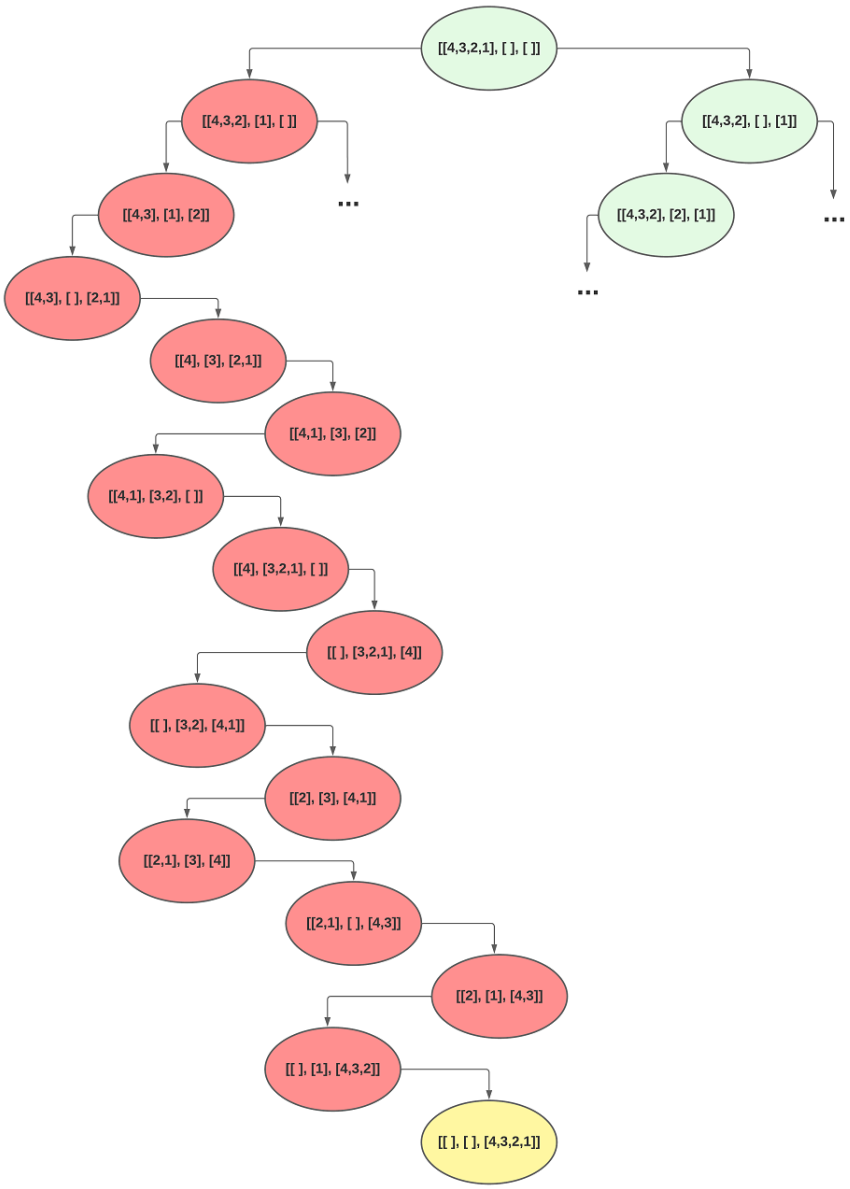
En caso de no encontrar una solución, la función del problema retorna “False” y un arreglo vació para luego indicar que no se encontró solución.

4. Reporte de la trayectoria

La trayectoria se da empezando con el estado inicial que es [[4,3,2,1], [ ], [ ]] que es la siguiente:

```
0) [[[4, 3, 2, 1], [ ], [ ]]
1) [[4, 3, 2], [1], [ ]]
2) [[4, 3], [1], [2]]
3) [[4, 3], [ ], [2, 1]]
4) [[4], [3], [2, 1]]
5) [[4, 1], [3], [2]]
6) [[4, 1], [3, 2], [ ]]
7) [[4], [3, 2, 1], [ ]]
8) [[ ], [3, 2, 1], [4]]
9) [[ ], [3, 2], [4, 1]]
10) [[2], [3], [4, 1]]
11) [[2, 1], [3], [4]]
12) [[2, 1], [], [4, 3]]
13) [[2], [1], [4, 3]]
14) [[ ], [1], [4, 3, 2]]
15) [[ ], [ ], [4, 3, 2, 1]]]
```

Para una mejor visualización se presenta la siguiente imagen:



5. Análisis de complejidad

En cuanto a la complejidad del algoritmo, hay que tomar en cuenta algunas cosas. Primero, la arborescencia es de 3. Como solo hay un máximo de 3 torres en el problema de las Torres de Hanói, si analizamos los posibles estados sucesores, veremos qué máximo solo puede haber 3 sucesores (El disco más pequeño solo puede moverse a las otras dos torres y si hay otro disco que se puede mover, este solo podrá moverse hacia un lado). Segundo, la profundidad dependerá del número de discos que hay, el cual será 'n' discos. Entendido lo anterior, tenemos la siguiente complejidad:

$$
\begin{gathered}
O(r^n y) \\
O(r^m) \text{ donde } m = n + \log_r y \\
n \log_r y = n + \log_3 1 = n \\
O(r^m) = O(3^n)
\end{gathered}
$$

6. Apéndices

6.1. Código del programa

```python
def func_objetivo(estado: list[list[int]], objetivo: list[list[int]]) -> bool:
    """
    Revisa que el estado sea el objetivo

    :param estado: Estado a revisar
    :param objetivo: Estado objetivo
    :return: True si el estado es el objetivo, False si no lo es
    """

    return estado == objetivo
```

Figura 1: Función Objetivo

```python
def func_sucesora(estado: list[list[int]]) -> list[list[list[int]]]:
    """
    Función que genera los sucesores del estado dado

    :param estado: Estado a revisar
    :return: Lista resultante de sucesores del estado dado
    """

    hijos = []

    for indice, torre in enumerate(estado):
        if torre:
            disco = torre.pop()
            for indice2, torre2 in enumerate(estado):
                if indice != indice2:  # No se puede mover a la misma torre
                    copia_estado = [i.copy() for i in estado]  # Se crea una copia del estado
                    if torre2 and torre2[-1] < disco:  # No se puede colocar un disco más grande sobre uno más pequeño
                        continue
                    copia_estado[indice2].append(disco)
                    hijos.append(copia_estado)
            torre.append(disco)

    return hijos
```

Figura 2: Función Sucesor

```python
def func_heuristica(estado: list[list[int]], objetivo: list[list[int]]) -> int:
    """
    Función heurística que calcula el número de discos que no están en su posición correcta

    :param estado: Estado a revisar
    :param objetivo: Estado objetivo
    :return: Número de discos que no están en su posición correcta
    """

    h = 0
    for i in range(len(estado)):
        for j in range(len(estado[i])):
            if j >= len(objetivo[i]) or estado[i][j] != objetivo[i][j]:  # Si el disco no está en su posición correcta
                h += 1
    return h
```

Figura 3: Función Heurística

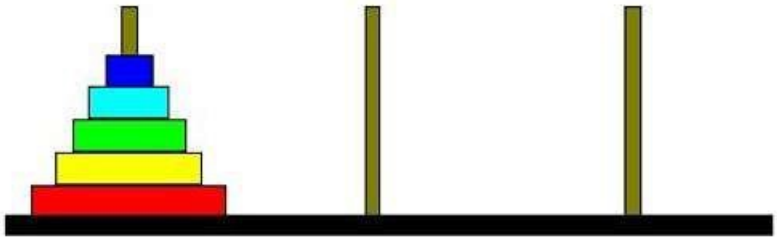
```python
def torres_hanoi(estado_inicial: list[list[int]], estado_final: list[list[int]]) -> (bool, list[list[int]]):
    """
    Función principal del problema de las torres de hanoi

    :param estado_inicial: Estado inicial del problema
    :param estado_final: Estado final del problema
    :return: True si se encontró una solución, False si no se encontró
    """

    frontera = [(estado_inicial, 0, None)]  # Lista de tuples (estado, costo, padre)
    visitados = {}

    while frontera:
        estado, costo, padre = frontera.pop(0)

        if str(estado) in visitados:
            continue
        visitados[str(estado)] = padre

        if func_objetivo(estado, estado_final):
            camino = [estado]
            while padre is not None:
                camino.append(padre)
                padre = visitados[str(padre)]
            camino.reverse()
            return True, camino

        sucesores = func_sucesora(estado)
        for sucesor in sucesores:
            costo_sucesor = costo + 1

            # Si el sucesor no está en la frontera ni en visitados
            if sucesor not in frontera and str(sucesor) not in visitados:
                frontera.append((sucesor, costo_sucesor, estado))

        frontera.sort(key=lambda x: x[1] + func_heuristica(x[0], estado_final))

    return False, []
```

Figura 4: Función torres_hanoi

```python
if __name__ == '__main__':
    from GUI import HanoiApp
    app = HanoiApp()
    app.algorithm = torres_hanoi
    app.run()
```

Figura 5: Función main

6.2. Manual de usuario



1. **Instalación:**

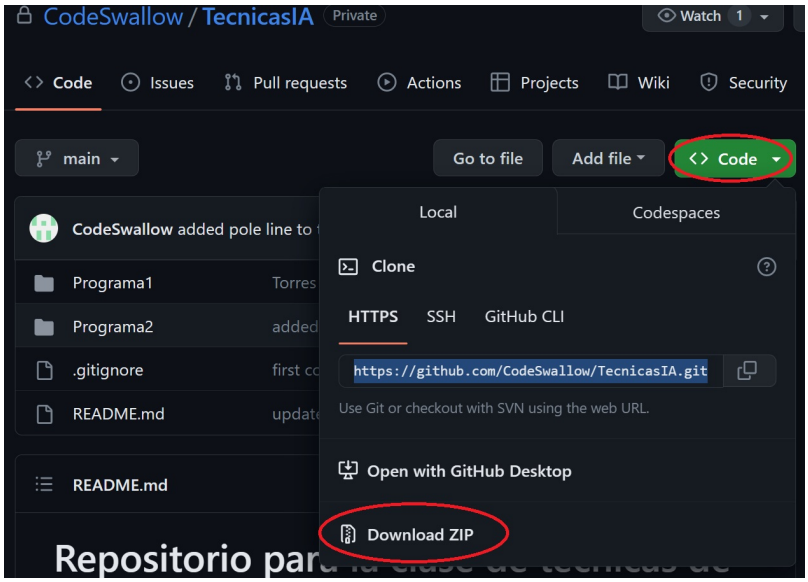
   a) Instalar Python desde su sitio oficial:
   https://www.python.org/downloads/

   b) Instalar/Actualizar pip, setuptools y vitualenv:
   python -m pip install –upgrade pip setuptools virtualenv

   c) Crear una carpeta donde guardar el programa y abrir terminal en ese directorio.

   d) Descargar el programa en la carpeta usando Git:
   git clone https://github.com/CodeSwallow/TecnicasIA.git

Si no se cuenta con Git entonces descargar archivo .zip del proyecto desde github.com, y descomprimirlo en la carpeta que creo anteriormente:
https://github.com/CodeSwallow/TecnicasIA



e) Crear un entorno virtual:
python -m virtualenv kivy_venv

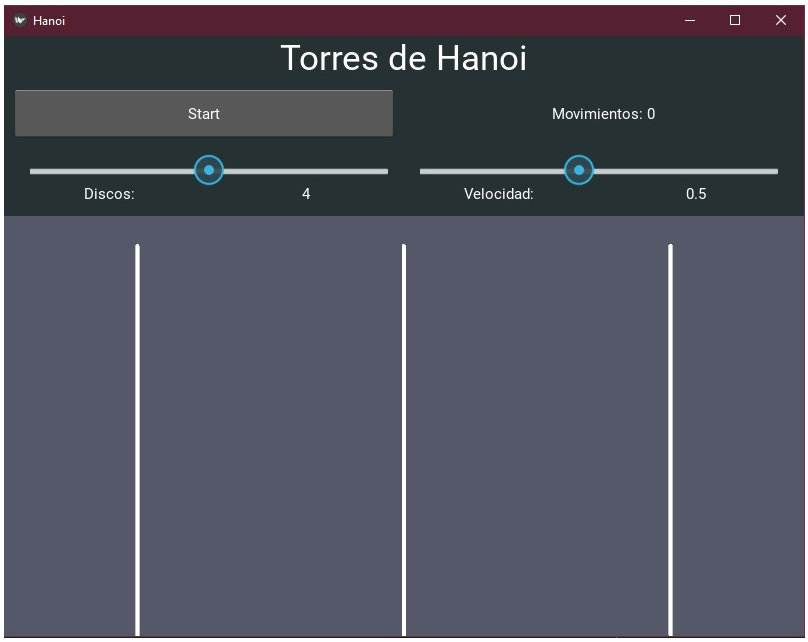
f) Activar el entorno virtual:
kivy_venv\Scripts\activate

g) Instalar los requerimientos:
pip install -r requirements.txt

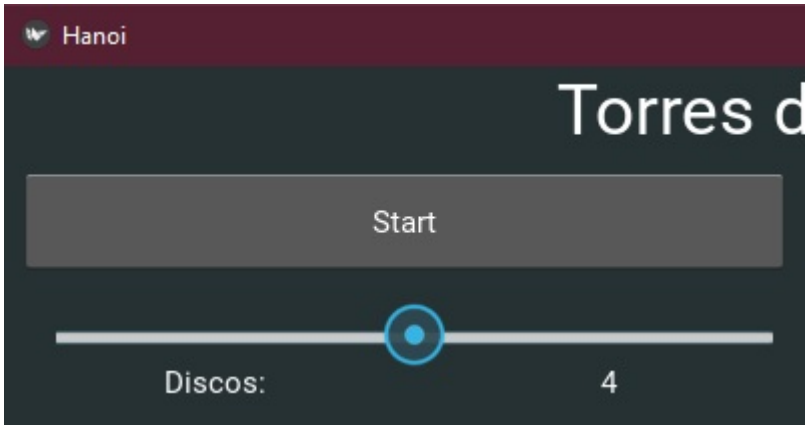
2. **Instrucciones:**

a) Ejecutar el programa con el comando:
python main.py

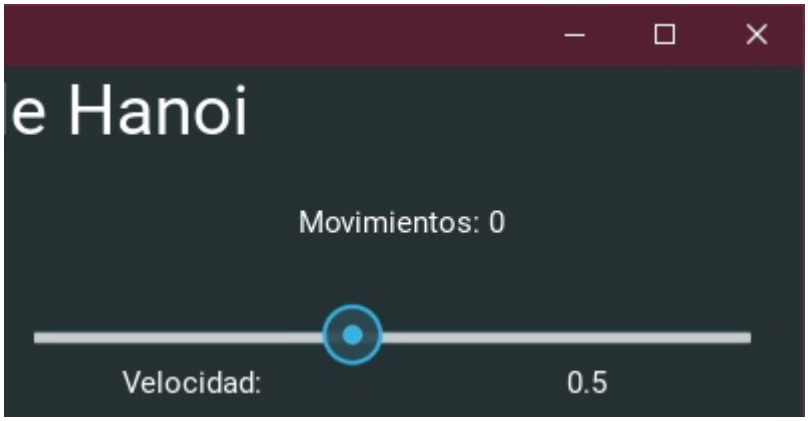
b) Al ejecutarse se tendra:



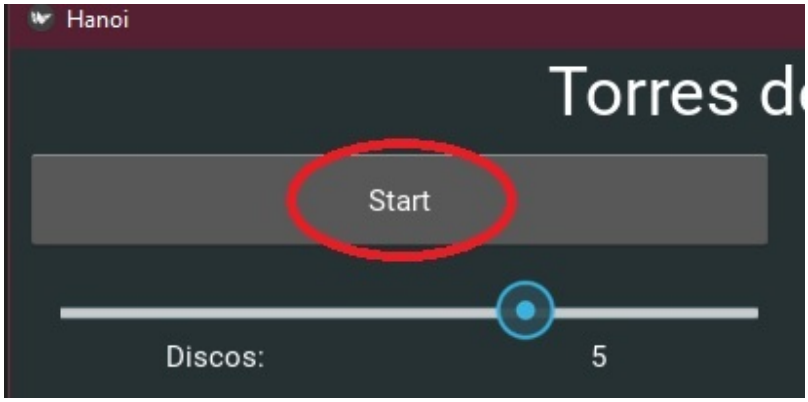
c) Control de número de disco: Para controlar la cantidad de discos, ajuste la barra, deslizándola de izquierda a derecha con el ratón. La barra va desde 1 hasta 7.



d) Control de velocidad: Para ajustar la velocidad, mueva la barra a la derecha para alentar la secuencia o mueva la barra a la izquierda para acelerar la secuencia.



e) Botón de inicio: Para iniciar, presione el botón Start.



Al iniciar, se representa como se resuelve la torre de Hanói paso a paso, actualizándose el número de movimientos a medida que la secuencia avanza.

Al final, la torre de Hanói cumple con su objetivo de colocar todos los discos en el poste objetivo en su orden correcto y se muestra la cantidad de movimientos requeridos para completar el juego.