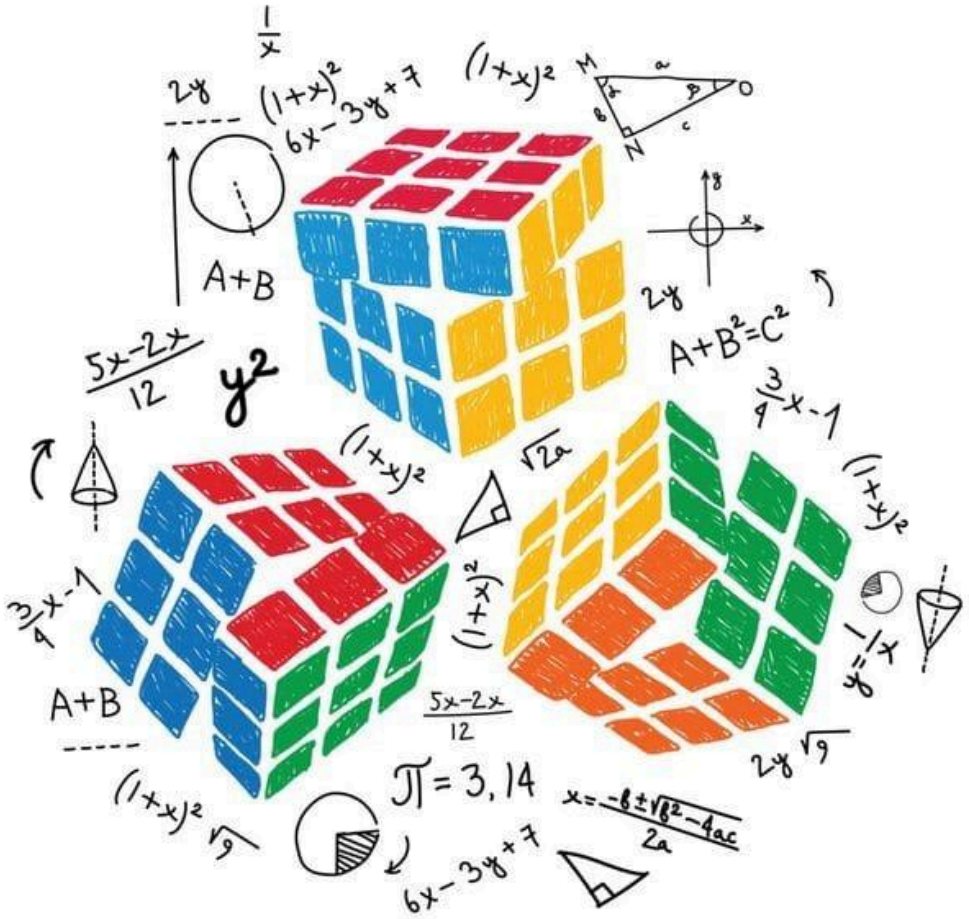
BÚSQUEDA EN ESPACIOS DE ESTADOS



CUBO DE RUBIK

Adán González Rodríguez **adan.rozaguez@gmail.com**
Santiago Palomino Vazquez **santivzqzz@gmail.com**

ÍNDICE

Adán González Rodríguez
Santiago Palomino Vazquez

Resumen del problema

Nuestra tarea es encontrar la manera de cambiar la configuración de un Cubo de Rubik (3x3) desde un estado desordenado hasta alcanzar el estado en el que todas sus caras estén completamente resueltas. Para lograrlo, necesitamos emplear algoritmos de búsqueda que nos ayuden a encontrar la secuencia de movimientos adecuada que nos lleve a la solución deseada.

Introducción

Esta práctica se enfoca en resolver el Cubo de Rubik mediante algoritmos de búsqueda y funciones heurísticas. El objetivo es explorar cómo estos métodos nos pueden llevar desde un estado cualquiera inicial hasta la solución óptima, considerando la complejidad del problema.

Por esta misma razón implementamos y probamos una variedad de algoritmos y heurísticas para determinar cuál funciona mejor en esta situación. Al explorar diferentes enfoques, podremos evaluar su eficacia y eficiencia en la resolución del Cubo de Rubik, ayudándonos a identificar qué método es más adecuado para alcanzar nuestro objetivo de manera óptima.

Descripción de Heurísticas

Heurística de Posiciones Correctas

Esta heurística calcula la cantidad de casillas que están en su posición correcta en el cubo de Rubik. Para cada cara del cubo, examina cada casilla y compara su color con el color de la cara. Si el color de la casilla coincide con el color de la cara, se incrementa el contador de la heurística. Esto significa que la casilla está en la posición correcta en relación con la cara correspondiente.

El objetivo de esta heurística es proporcionar una estimación del progreso hacia la solución del cubo. Cuantas más casillas estén en su posición correcta, más cerca estará el cubo de estar resuelto. Sin embargo, esta heurística no tiene en cuenta la orientación de las casillas, por lo que puede no ser suficiente para resolver el cubo en todos los casos.

```
def heuristicaPosicionesCorrectas(self):
    heuristica = 0
    for cara in self.caras:
        for casilla in cara.casillas:
            if casilla.posicionCorrecta == cara.color:
                heuristica += 1
    return heuristica
```

Heurística de Casillas Descolocadas

Esta heurística cuenta la cantidad de casillas que están descolocadas en el cubo de Rubik. Para cada cara del cubo, itera sobre cada casilla y compara su color con el color de la cara. Si el color de la casilla no coincide con el color de la cara, se incrementa el contador de la heurística. Esto indica que la casilla no está en la posición correcta en relación con la cara correspondiente.

El propósito de esta heurística es proporcionar una medida del nivel de desorden o "confusión" del cubo. Cuantas más casillas estén descolocadas, más difícil será resolver el cubo. Esta heurística tiene en cuenta tanto la posición como la orientación de las casillas, lo que puede ayudar a guiar el proceso de resolución hacia una configuración más ordenada del cubo.

```
def heuristicaCasillasDescolocadas(self):
    heuristica = 0
    for cara in self.caras:
        for casilla in cara.casillas:
            if casilla.color != cara.color:
                heuristica += 1
    return heuristica
```

Cabe destacar que ambas heurísticas están implementadas en *cubo.py*

Métodos

Búsqueda en Anchura

El algoritmo comienza con una lista de nodos abiertos, que contiene el nodo inicial. Se mantiene también un diccionario para los nodos cerrados, que almacena los nodos ya visitados. Durante el proceso de búsqueda, se extraen nodos de la lista de abiertos, se comprueba si es la solución deseada y, en caso contrario, se expande ese nodo y se añaden sus hijos a la lista de abiertos si no han sido visitados antes.

Si se encuentra una solución, se reconstruye el camino desde el nodo final hasta el inicial. Se devuelven los pasos del camino y estadísticas de la búsqueda, como el número total de nodos explorados y el tamaño máximo de la lista de abiertos.

```
class BusquedaAnchura(Busqueda):
    def buscarSolucion(self, inicial):
        nodoActual = None
        actual, hijo = None, None
        solucion = False
        abiertos = []
        cerrados = dict()

        num_nodos_explorados = 0
        max_tam_abiertos = len(abiertos)

        abiertos.append(NodoAnchura(inicial, None, None))
        cerrados[inicial.cubo.visualizar()] = inicial

        while not solucion and len(abiertos) > 0:
            nodoActual = abiertos.pop(0)
            actual = nodoActual.estado
            num_nodos_explorados += 1
            max_tam_abiertos = max(max tam abiertos, len(abiertos))

            if actual.esFinal():
                solucion = True
            else:
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    if hijo.cubo.visualizar() not in cerrados.keys():
                        abiertos.append(NodoAnchura(hijo, nodoActual, operador))
                        cerrados[hijo.cubo.visualizar()] = hijo
        if solucion:
            lista = []
            nodo = nodoActual
            while nodo.padre is not None:
                lista.insert(0, nodo.operador)
                nodo = nodo.padre
            resultado = [f"Número total de nodos explorados: {num_nodos_explorados}",
                         f"Tamaño máximo de la lista ABIERTOS: {max tam abiertos}",
                         f"Tamaño de la solución encontrada: {len(lista)}"]
            return lista, resultado
        else:
            return None
```

Búsqueda en Profundidad

Al igual que en la búsqueda en anchura, se comienza con una lista de nodos abiertos, que contiene el nodo inicial, y un conjunto de nodos cerrados para evitar la revisión de nodos ya visitados. Durante la búsqueda, se extrae el último nodo de la lista de abiertos, se comprueba si es la solución deseada y, si no lo es y no se ha alcanzado la profundidad máxima, se expande el nodo actual y se añaden sus hijos a la lista de abiertos.

Si se encuentra una solución, se reconstruye el camino desde el nodo final hasta el inicial y se devuelven los pasos del camino y estadísticas de la búsqueda.

```python
class BusquedaProfundidad(Busqueda):
    def buscarSolucion(self, inicial, profundidad_maxima=float('inf')):
        nodoActual = None
        actual, hijo = None, None
        solucion = False
        abiertos = []
        cerrados = set()

        num_nodos_explorados = 0
        max_tam_abiertos = len(abiertos)

        abiertos.append(NodoProfundidad(inicial, None, None, 0))
        cerrados.add(inicial.cubo.visualizar())

        while not solucion and len(abiertos) > 0:
            nodoActual = abiertos.pop()
            actual = nodoActual.estado
            num_nodos_explorados += 1
            max_tam_abiertos = max(max_tam_abiertos, len(abiertos))

            if actual.esFinal():
                solucion = True
            elif nodoActual.profundidad < profundidad_maxima:
                cerrados.add(actual.cubo.visualizar())
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    if hijo.cubo.visualizar() not in cerrados:
                        abiertos.append(NodoProfundidad(hijo, nodoActual, operador, nodoActual.profundidad + 1))
        if solucion:
            lista = []
            while nodoActual.padre is not None:
                lista.insert(0, nodoActual.operador)
                nodoActual = nodoActual.padre
            resultado = [f"Número total de nodos explorados: {num_nodos_explorados}",
                         f"Tamaño máximo de la lista ABIERTOS: {max_tam_abiertos}",
                         f"Tamaño de la solución encontrada: {len(lista)}"]
            return lista, resultado
        else:
            return None
```

Búsqueda en Profundidad Iterativa

El algoritmo realiza iteraciones de búsqueda en profundidad con una profundidad máxima inicial. Si no se encuentra una solución en una iteración, se repite la búsqueda con una profundidad máxima incrementada en la siguiente iteración. Esto continúa hasta que se encuentra una solución.

Una vez se halla una solución, se devuelve el camino y las estadísticas de la búsqueda.

```
class BusquedaProfundidadIterativa(Busqueda):
    def buscarSolucion(self, inicial):
        profundidad_maxima = 0
        solucion = None
        while solucion is None:
            solucion = BusquedaProfundidad().buscarSolucion(inicial, profundidad_maxima)
            profundidad_maxima += 1
        return solucion
```

Búsqueda Voraz

La búsqueda comienza con una lista de nodos abiertos, que contiene el nodo inicial, y un conjunto de nodos cerrados para evitar la revisión de nodos ya visitados. Durante la búsqueda, se selecciona el nodo con la menor heurística de la lista de abiertos, se expande y se añaden sus hijos a la lista de abiertos si no han sido visitados antes.

Si se encuentra una solución, se reconstruye el camino desde el nodo final hasta el inicial y se devuelven los pasos del camino y estadísticas de la búsqueda.

```
class BusquedaVoraz(Busqueda):
    def buscarSolucion(self, inicial):
        abiertos = [NodoVoraz(inicial, None, None, 0)]
        cerrados = set()
        solucion = False
        num_nodos_explorados = 0
        max_tam_abiertos = len(abiertos)
        while abiertos and not solucion:
            abiertos.sort(key=lambda x: x.heuristica)
            nodo_actual = abiertos.pop(0)
            actual = nodo_actual.estado
            num_nodos_explorados += 1
            max_tam_abiertos = max(max_tam_abiertos, len(abiertos))
            if actual.esFinal():
                solucion = True
            else:
                cerrados.add(actual.cubo.visualizar())
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    heuristica = hijo.cubo.heuristicaCasillasDescolocadas()
                    if hijo.cubo.visualizar() not in cerrados:
                        abiertos.append(NodoVoraz(hijo, nodo_actual, operador, heuristica))
        if solucion:
            lista = []
            while nodo_actual.padre:
                lista.insert(0, nodo_actual.operador)
                nodo_actual = nodo_actual.padre
            resultado = [f"Número total de nodos explorados: {num_nodos_explorados}",
                         f"Tamaño máximo de la lista ABIERTOS: {max_tam_abiertos}",
                         f"Tamaño de la solución encontrada: {len(lista)}"]
            return lista, resultado
        else:
            return None
```

Búsqueda A*

Al igual que en los otros algoritmos, se inicia con una lista de nodos abiertos, que contiene el nodo inicial, y un conjunto de nodos cerrados. Durante la búsqueda, se selecciona el nodo con el menor valor de la suma del costo acumulado y la heurística de la lista de abiertos, se expande y se añaden sus hijos a la lista de abiertos si no han sido visitados antes.

Si se encuentra una solución, se reconstruye el camino desde el nodo final hasta el inicial y se devuelven los pasos del camino y estadísticas de la búsqueda.

```
class BusquedaAStar(Busqueda):
    def buscarSolucion(self, inicial):
        abiertos = [NodoAStar(inicial, None, None, 0, 0)]
        cerrados = set()
        solucion = False

        num_nodos_explorados = 0
        max_tam_abiertos = len(abiertos)

        while abiertos and not solucion:
            abiertos.sort(key=lambda x: x.costo + x.heuristica)
            nodo_actual = abiertos.pop(0)
            actual = nodo_actual.estado
            num_nodos_explorados += 1
            max_tam_abiertos = max(max_tam_abiertos, len(abiertos))

            if actual.esFinal():
                solucion = True
            else:
                cerrados.add(actual.cubo.visualizar())
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    heuristica = hijo.cubo.heuristicaCasillasDescolocadas()
                    costo_acumulado = nodo_actual.costo + 1
                    if hijo.cubo.visualizar() not in cerrados:
                        abiertos.append(NodoAStar(hijo, nodo_actual, operador, costo_acumulado, heuristica))
        if solucion:
            lista = []
            while nodo_actual.padre:
                lista.insert(0, nodo_actual.operador)
                nodo_actual = nodo_actual.padre
            resultado = [f"Número total de nodos explorados: {num_nodos_explorados}",
                         f"Tamaño máximo de la lista ABIERTOS: {max_tam_abiertos}",
                         f"Tamaño de la solución encontrada: {len(lista)}"]

            return lista, resultado
        else:
            return None
```

Búsqueda IDA*

El algoritmo comienza con un cálculo de la estimación inicial para la heurística de posiciones correctas del nodo inicial. Se establece un límite inicial en esta estimación. Luego, se realiza una búsqueda en profundidad limitada utilizando la heurística de posiciones correctas y un límite progresivamente incrementado. Esto continúa hasta que se encuentra una solución.

Una vez se halla una solución, se devuelve el camino y las estadísticas de la búsqueda.

```
class BusquedaIDAStar(Busqueda):
    def __init__(self):
        self.heuristica_idastar = None

        self.num_nodos_explorados = 0
        self.max_tam_abiertos = 0

    def buscarSolucion(self, inicial):
        self.heuristica_idastar = self.calcularHeuristica(inicial)
        limite = self.heuristica_idastar
        solucion = None

        while solucion is None:
            solucion, limite = self.buscarSolucionRecursiva(inicial, 0, limite)

            resultado = [f"Número total de nodos explorados: {self.num_nodos_explorados}",
                         f"Tamaño máximo de la lista ABIERTOS: {self.max_tam_abiertos}",
                         f"Tamaño de la solución encontrada: {len(solucion if solucion else [])}"]

        return solucion, resultado

    def buscarSolucionRecursiva(self, estado, costo_acumulado, limite):
        heuristica = self.calcularHeuristica(estado)
        f = costo_acumulado + heuristica

        self.num_nodos_explorados += 1
        self.max_tam_abiertos = max(self.max_tam_abiertos, len(estado.operadoresAplicables()))

        if f > limite:
            return None, f

        if estado.esFinal():
            return [], f

        minimo = float('inf')
        for operador in estado.operadoresAplicables():
            hijo = estado.aplicarOperador(operador)
            costo = 1
            resultado, nuevo_limite = self.buscarSolucionRecursiva(hijo, costo_acumulado + costo, limite)
            if resultado is not None:
                resultado.insert(0, operador)
                return resultado, nuevo_limite
            if nuevo_limite < minimo:
                minimo = nuevo_limite

        return None, minimo

    def calcularHeuristica(self, estado):
        heuristica = estado.cubo.heuristicaPosicionesCorrectas()
        return heuristica
```

Modificaciones

El archivo *main.py* ha sido modificado para permitir la ejecución de múltiples algoritmos uno tras otro, con la opción de manejar errores de manera que la ejecución no se detenga por completo en caso de un error en un algoritmo. Además, se ha agregado la funcionalidad de guardar los datos de la ejecución en un archivo de texto '*tiempos.txt*'

```
import sys
from cubo import *
from problemaRubik import *
from busqueda import *
import time

cubo = Cubo()
with open('tiempos.txt', 'a', encoding='utf-8') as f:
    sys.stdout = f
    print("CUBO SIN MEZCLAR:\n" + cubo.visualizar())


    #Mover frontal face
    cubo.mover(cubo.F)

    print("CUBO resultado del movimiento F:\n" + cubo.visualizar())

    movs=int(sys.argv[1])

    movsMezcla = cubo.mezclar(movs)

    print("MOVIMIENTOS ALEATORIOS:",movs)
    for m in movsMezcla:
        print(cubo.visualizarMovimiento(m) + " ")
    print()

    print("CUBO INICIAL (MEZCLADO):\n" + cubo.visualizar())
    print('------------------------------------\n')
    sys.stdout = sys.__stdout__

algoritmos = [BusquedaAnchura(), BusquedaProfundidadIterativa(), BusquedaVoraz(), BusquedaAStar(), BusquedaIDAStar(),
BusquedaProfundidad()]

for i in range(len(algoritmos)):
    cubo = Cubo()
    seed(14)
    problema = Problema(EstadoRubik(cubo), algoritmos[i])
    print(f"ALGORITMO: {algoritmos[i].__class__.__name__} [EJECUTANDO]")
    #Mover frontal face
    cubo.mover(cubo.F)

    movs=int(sys.argv[1])

    movsMezcla = cubo.mezclar(movs)

    a = time.time()
```

```
    try:
        with open('tiempos.txt', 'a', encoding='utf-8') as f:
            sys.stdout = f
            print()
            print(f"ALGORITMO: {algoritmos[i].__class__.__name__}\t nºmovs: {movs}\n")
            print("SOLUCION:")
            opsSolucion = problema.obtenerSolucion()
            if opsSolucion != None:
                for o in opsSolucion[0]:
                    print(cubo.visualizarMovimiento(o.getEtiqueta()) + " ")
                    cubo.mover(o.movimiento)
                print()
                print("CUBO FINAL:\n" + cubo.visualizar())
                b = time.time()
                print(f"TIEMPO: {b-a}\n")
                for r in opsSolucion[1]:
                    print(r)
                print()
            else:
                print("no se ha encontrado solución")
                b = time.time()
                print(f"TIEMPO: {b-a}\n")

            print('-------------------------------------')
            sys.stdout = sys.__stdout__

        print(f"ALGORITMO: {algoritmos[i].__class__.__name__} [FINALIZADO]\n")

    except KeyboardInterrupt:
        b = time.time()
        sys.stdout = sys.__stdout__
        print(f"\nLa ejecución ha sido interrumpida.\nTIEMPO: {b-a}\n")
        with open('tiempos.txt', 'a', encoding='utf-8') as f:
            f.write(f"\nLa ejecución ha sido interrumpida.\nTIEMPO: {b-a}\n")
            f.write('\n-------------------------------------\n')

    except Exception as e:
        b = time.time()
        sys.stdout = sys.__stdout__
        print(f"\nHa ocurrido un error: {str(e)}.\nTIEMPO: {b-a}\n")
        with open('tiempos.txt', 'a', encoding='utf-8') as f:
            f.write(f"\nHa ocurrido un error: {str(e)}.\nTIEMPO: {b-a}\n")
            f.write('\n-------------------------------------\n')
```

Por otra parte, hemos creado un nodo para cada algoritmo en el archivo *nodos.py* para evitar errores al reutilizar clases nodo que pueden no funcionar para algún algoritmo en específico.

Criterios de Evaluación

Para evaluar los distintos algoritmos hemos decidido fijarnos en los siguientes parámetros:

El tiempo de resolución nos sirve para calcular la eficiencia temporal del algoritmo, en este caso nos interesa saber cuál tarda menos. Hemos establecido de tiempo máximo de ejecución 2 horas, por esta razón algoritmos como el IDA* que en 8 movimientos ha acabado en 2h23min lo hemos descartado de los resultados obtenidos.

El número de Nodos Explorados nos sirve para saber lo eficiente que ha sido la búsqueda en los algoritmos y de esta forma evaluar nuestras heurísticas.

El tamaño de la Lista de Abiertos nos permite saber que algoritmo consume menos recursos.

El **resultado obtenido** nos sirve para saber si el algoritmo ha dado la solución óptima o no.

Resultados Obtenidos

Comentaremos los resultados obtenidos con los experimentos realizados en el ordenador 1. Más adelante se muestran las tablas con todos los tiempos de los 2 ordenadores donde se han realizado dichos experimentos.

5 Movimientos

En todos los casos debido a los movimientos de la semilla (14) usada la solución es: U Fi

Tiempo de Resolución

Los tiempos de resolución varían entre los diferentes algoritmos, siendo los más rápidos la **Búsqueda Voraz, Búsqueda A* y Búsqueda IDA*** con tiempos de alrededor de 0.0015 segundos, seguidos por la **Búsqueda en Profundidad Iterativa** con 0.006 segundos, y finalmente la **Búsqueda en Anchura** con 0.0095 segundos. Esto muestra que los algoritmos más informados logran encontrar la solución más eficientemente en comparación con los algoritmos que no están siendo guiados.

Número de Nodos Explorados

La cantidad de nodos explorados varía significativamente entre los algoritmos. La **Búsqueda en Anchura** y la **Búsqueda en Profundidad Iterativa** exploran un número considerablemente mayor de nodos en comparación con los otros algoritmos, con 18 y 141 nodos respectivamente. En contraste, los algoritmos informados exploran solo 3 nodos cada uno para encontrar la solución. Esto demuestra la eficacia de los algoritmos informados en la reducción del número de nodos explorados.

Tamaño de la Lista Abiertos

La **Búsqueda en Profundidad Iterativa** muestra el tamaño máximo más bajo de la lista **ABIERTOS** (21), mientras que la **Búsqueda en Anchura** muestra el más alto (151). Esto refleja la naturaleza de estos algoritmos, donde la **Búsqueda en Anchura** tiende a expandir todos los nodos en cada nivel antes de pasar al siguiente, lo que resulta en una lista **ABIERTOS** más grande, mientras que la **Búsqueda en Profundidad Iterativa** mantiene una lista **ABIERTOS** más pequeña ya que solo expande un nodo a la vez en profundidad.

6 Movimientos

En todos los casos debido a los movimientos de la semilla (14) usada la solución es: L U Fi

Tiempo de Resolución

Los tiempos de resolución varían entre los diferentes algoritmos, con la **Búsqueda IDA*** mostrando el tiempo más largo de resolución (14.86 segundos), seguido por la **Búsqueda en Anchura** (0.234 segundos), **Búsqueda en Profundidad Iterativa** (0.040 segundos), **Búsqueda A*** (0.0015 segundos) y **Búsqueda Voraz** (0.0014 segundos).

Número de Nodos Explorados

La cantidad de nodos explorados varía significativamente entre los algoritmos. **La Búsqueda IDA*** muestra el mayor número de nodos explorados (679671), seguido por la **Búsqueda en Anchura** (328 nodos) y la **Búsqueda en Profundidad Iterativa** (1171 nodos).

Tamaño de la Lista Abiertos

La **Búsqueda IDA*** muestra el tamaño máximo más bajo de la lista **ABIERTOS** (12), mientras que la **Búsqueda en Anchura** muestra el más alto (2746). Esto sigue reflejando las diferencias entre los algoritmos, donde la **Búsqueda en Anchura** expande todos los nodos en cada nivel antes de pasar al siguiente, lo que resulta en una lista **ABIERTOS** más grande, mientras que la **Búsqueda en Profundidad Iterativa** mantiene una lista **ABIERTOS** más pequeña. Se observa que la **Búsqueda IDA*** requiere mucho más tiempo y nodos explorados en comparación con los otros algoritmos informados, lo que sugiere que, en este caso específico, la **Búsqueda IDA*** no es tan eficiente.

7 Movimientos

En todos los casos debido a los movimientos de la semilla (14) usada la solución es: Fi L U Fi

Tiempo de Resolución

La Búsqueda Profundidad Iterativa es la más rápida, con un tiempo de resolución de 0.335 segundos, seguida por la **Búsqueda en Anchura** con 6.588 segundos, **Búsqueda A*** con 13.552 segundos y finalmente la **Búsqueda IDA*** con un tiempo de resolución de 397.271 segundos. Esto muestra una clara diferencia en la eficiencia de los algoritmos.

Número de Nodos Explorados

El número de nodos explorados varía considerablemente entre los algoritmos. La **Búsqueda IDA*** muestra el mayor número de nodos explorados con 16315353, seguida por la **Búsqueda en Anchura** con 6035 nodos, Búsqueda A* con 3686 nodos y la **Búsqueda en Profundidad Iterativa** con 8033 nodos explorados.

Tamaño de la Lista Abiertos

La **Búsqueda Profundidad Iterativa** muestra el tamaño máximo más bajo de la lista **ABIERTOS** (40), mientras que la **Búsqueda en Anchura** muestra el máximo más alto (50536).

8 Movimientos

En este caso la solución varía entre los distintos algoritmos, la profundidad Iterativa no consigue la solución Óptima.

Profundidad Iterativa obtiene: Di Di Di Fi L U Fi
A* Obtiene: D Fi L U Fi
Anchura Obtiene: D Fi L U Fi
El resto de algoritmos no acaban.

Tiempo de Resolución

La Búsqueda Profundidad Iterativa es la más rápida, con un tiempo de resolución de 11.943 segundos, seguida por la **Búsqueda en Anchura** con 112.041 segundos y finalmente la **Búsqueda A*** con un tiempo de resolución de 499.344 segundos. Esto muestra una clara diferencia en la eficiencia de los algoritmos.

Número de Nodos Explorados

La **Búsqueda en Anchura** muestra el mayor número de nodos explorados con 93989, seguida por la **Búsqueda A*** con 17238 nodos y la **Búsqueda en Profundidad Iterativa** con 11943 nodos explorados.

Tamaño de la Lista Abiertos

La **Búsqueda Profundidad Iterativa** muestra el tamaño máximo más bajo de la lista **ABIERTOS** (70), la **Búsqueda A***(168259) mientras que la **Búsqueda en Anchura** muestra el máximo más alto (786470).

+8 Movimientos

Ningún algoritmo acaba en el plazo establecido de 2h

Tiempos obtenidos

Ordenador 1

nº movimientos	1	2	3	4	5	6	7	8	9	10
Anchura	0,009	0,850	10,884	0,847	0,0095	0,23	6,59	122,04	Inf*	Inf*
Profundidad Iterativa	0,006	0,027	0,117	0,025	0,006	0,04	0,33	11,94	Inf*	Inf*
A*	0,001	0,002	1,008	0,002	0,0015	0,0015	13,55	499,34	Inf*	Inf*
IDA*	0,710	51,090	634,770	51,171	0,7	14,86	397,27	8623	Inf*	Inf*
Voraz	0,010	0,002	Inf*	0,002	0,0015	0,0014	Inf*	Inf*	Inf*	Inf*
Profundidad	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*

El tiempo está en segundos
*Inf → Indica que algoritmo no ha conseguido acabar con ese número de movimientos

Ordenador 2

nº movimientos	1	2	3	4	5	6	7	8	9	10
Anchura	0,009	0,819	11,310	0,815	0,0095	0,254	6,88	135,006	Inf*	Inf*
Profundidad Iterativa	0,006	0,026	0,113	0,025	0,0065	0,049	0,35	13,066	Inf*	Inf*
A*	0,001	0,002	0,649	0,002	0,001	0,0015	14,94	537.844	Inf*	Inf*
IDA*	0,723	51,083	817,023	51,166	0,717	17,43	418,37	9398,945	Inf*	Inf*
Voraz	0,010	0,002	Inf*	0,002	0,001	0,0015	Inf*	Inf*	Inf*	Inf*
Profundidad	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*

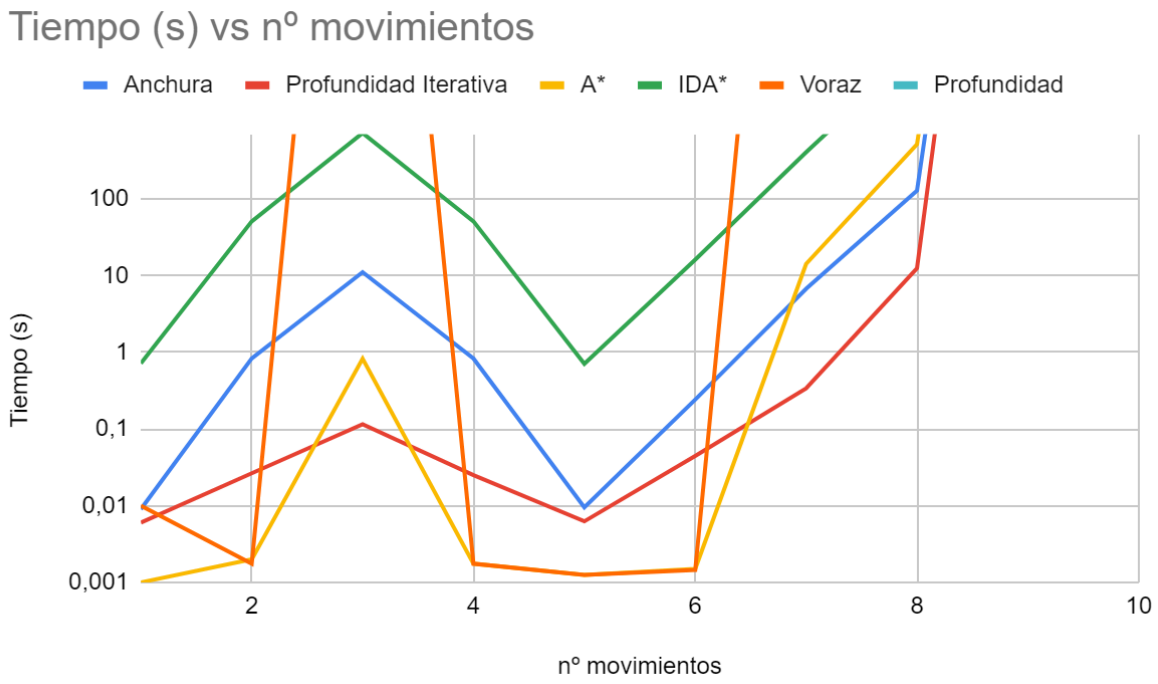
El tiempo está en segundos
*Inf → Indica que algoritmo no ha conseguido acabar con ese número de movimientos

Media entre ordenador 1 y ordenador 2

nº movimientos	1	2	3	4	5	6	7	8	9	10
Anchura	0,009	0,835	11,097	0,831	0,0095	0,242	6,735	128,523	Inf*	Inf*
Profundidad Iterativa	0,006	0,027	0,115	0,025	0,0063	0,045	0,34	12,503	Inf*	Inf*
A*	0,001	0,002	0,829	0,002	0,0013	0,002	14,245	518,593	Inf*	Inf*
IDA*	0,717	51,087	725,897	51,169	0,709	16,145	407,82	9010,97	Inf*	Inf*
Voraz	0,010	0,002	Inf*	0,002	0,0013	0,0015	Inf*	Inf*	Inf*	Inf*
Profundidad	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*	Inf*

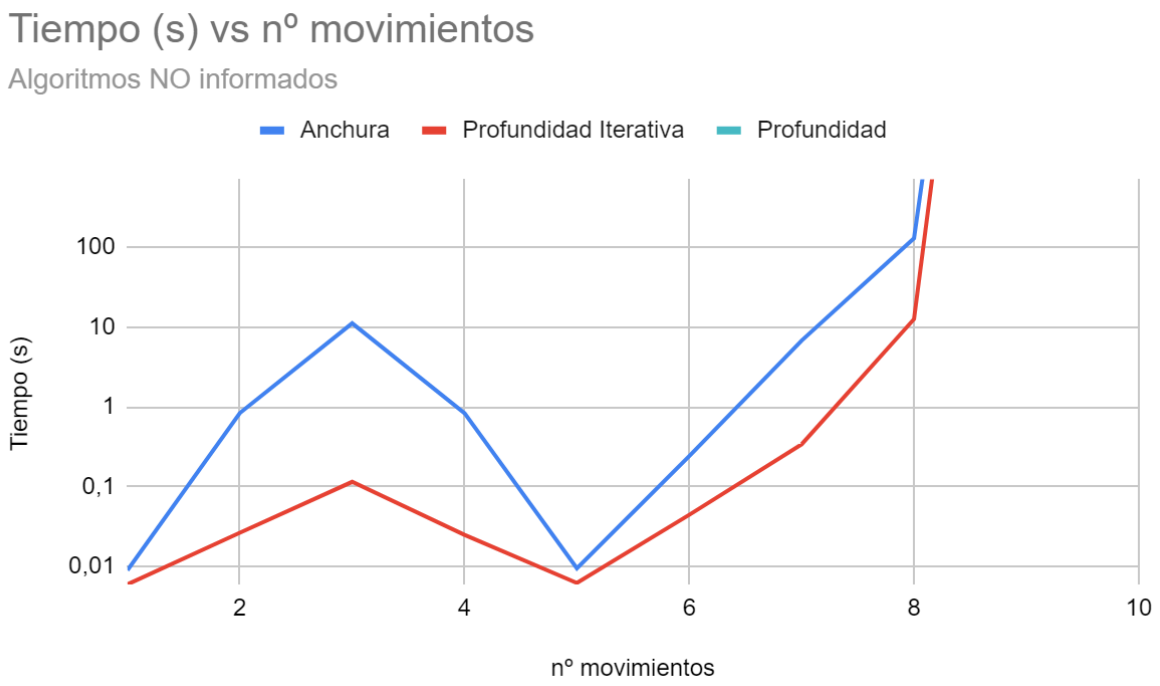
El tiempo está en segundos
*Inf → Indica que algoritmo no ha conseguido acabar con ese número de movimientos

Gráfica comparativa



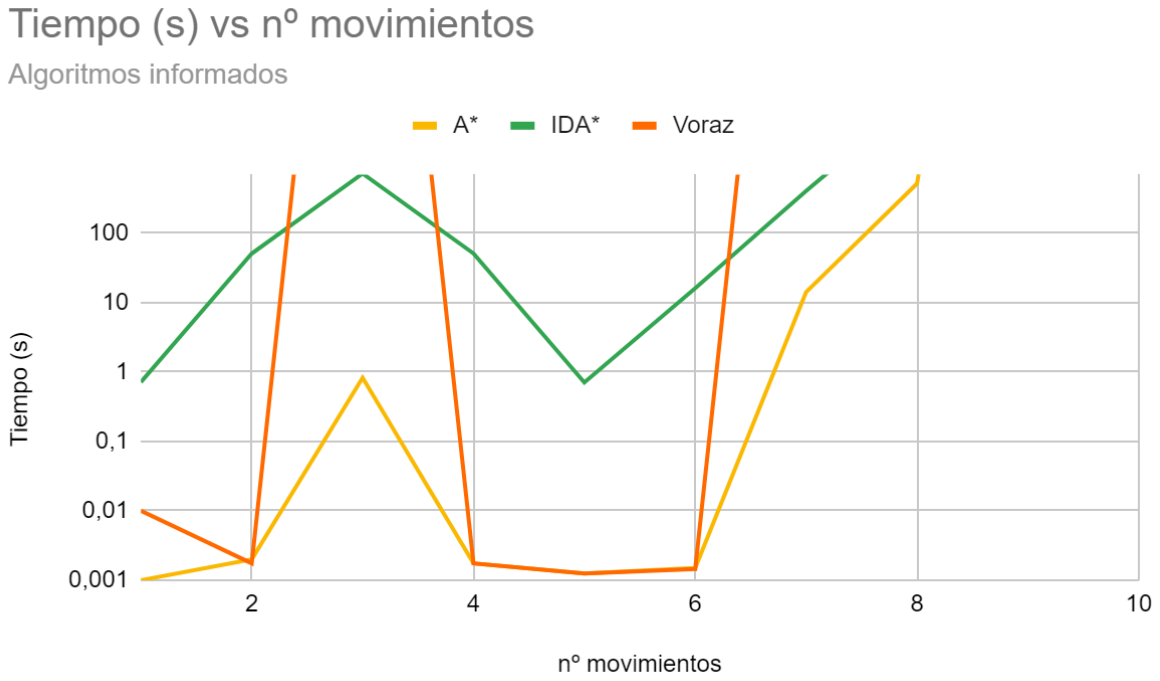
Gráfica realizada a partir de la tabla con la media de los tiempos entre los dos ordenadores.

Gráfica algoritmos no informados



Gráfica realizada a partir de la tabla con la media de los tiempos entre los dos ordenadores.

Gráfica algoritmos informados



Gráfica realizada a partir de la tabla con la media de los tiempos entre los dos ordenadores.

Manual de uso

Uso del programa

1. Entrar en la carpeta "**rubik**" desde terminal para la ejecución.
2. Escribir en la terminal "**python main.py X**" siendo X el número de movimientos deseado para la ejecución.
3. Si todo ha funcionado correctamente se imprimirá el resultado por pantalla.
4. En caso de que alguna de las ejecuciones no funcione pruebe de nuevo sustituyendo el "**python**" inicial de los comandos por "**python3**".

Conclusiones

Observando los resultados obtenidos podemos ver que de forma **temporal** el mejor es el algoritmo en **Profundidad Iterativa**, a pesar de esto debemos tener en cuenta que a partir de los 8 movimientos la solución encontrada no es la óptima. Por esta misma razón podemos determinar que para este problema en concreto, si queremos buscar la mayor eficiencia temporal usaremos el de **Profundidad Iterativa**, mientras que si quisiéramos buscar la solución óptima usaríamos el **A*** siempre y cuando dispongamos de la memoria necesaria que exige la **búsqueda A***.

Por otra parte, al utilizar la misma semilla para la generación de movimientos, los resultados tras cada ejecución eran los mismos, excepto cuando se ejecutaba en ordenadores distintos donde cambian los tiempos de ejecución en algunos algoritmos. Esto debe de ser por la diferencia en el hardware de cada ordenador.

Problemas

Durante el desarrollo del proyecto, nos enfrentamos a varios desafíos, siendo uno de los más significativos la dificultad para implementar la heurística de Manhattan. Esto nos llevó a utilizar heurísticas que podrían no ser tan eficientes como esperábamos. Además, encontramos dificultades al intentar resolver el cubo con un número limitado de movimientos, particularmente cuando se trataba de resolverlo con más de 8 movimientos.

Además, en la tabla de los datos y en las gráficas, se puede ver una subida considerable en el tiempo de obtener la solución cuando se trata de 3 movimientos y luego vuelve a bajar a partir de los 4. Esto puede deberse a que la semilla utilizada para desmontar el cubo revierte uno de los movimientos de este, lo que facilita su resolución.

En cuanto a la incapacidad para alcanzar una solución con más de ocho movimientos, podría deberse a limitaciones en el equipo utilizado para las pruebas o a la ineficiencia de las heurísticas implementadas. Es posible que las heurísticas utilizadas no proporcionen una guía adecuada para explorar eficientemente el espacio de búsqueda en estos casos más complejos.

Otro problema a destacar, es el algoritmo de búsqueda en profundidad que en ningún caso ha conseguido obtener una solución con ningún número de movimientos. Esto puede estar causado por la semilla utilizada o por la memoria disponible de los equipos.