Water Sort Puzzle

Eduardo González Bautista

Juan manuel Valenzuela González

A green liquid in a test tube

AI-generated content may be incorrect.

ÍNDICE

[1. Introducción 3](#_Toc211946479)

[2. Estructura del proyecto 3](#_Toc211946480)

[3. Arquitectura de la aplicación 4](#_Toc211946481)

[1. Representación de Estados 4](#_Toc211946482)

[2. Clase WaterSortGame 4](#_Toc211946483)

[3. Clase SearchAlgorithms 5](#_Toc211946484)

[4. Heurísticas Disponibles 6](#_Toc211946485)

[4.1 Entropy (Entropía) - DEFAULT 6](#_Toc211946486)

[4.2 Completion (Completitud) 6](#_Toc211946487)

[4.3 Blocking (Bloqueo) 7](#_Toc211946488)

[5. Análisis comparativo 8](#_Toc211946489)

[Resultados 8](#_Toc211946490)

[6. Casos de prueba 10](#_Toc211946491)

[7. Conclusión 13](#_Toc211946492)

# Introducción

Este proyecto implementa un solucionador automático para el **Water Sort Puzzle**, modelado como un problema de búsqueda en espacio de estados. Incluye múltiples algoritmos de búsqueda informada y no informada, junto con heurísticas configurables para optimizar el rendimiento.

# Estructura del proyecto

Practica evaluable 1/

│

├── watersort/

│ ├── \_\_init\_\_.py # Inicialización del módulo

│ ├── game.py # Lógica del juego y generación

│ ├── search.py # Algoritmos de búsqueda

│ └── heuristics.py # Funciones heurísticas

│

├── water\_sort\_solver.py # CLI principal

├── water\_sort\_gui.py # Interfaz gráfica Tkinter

└── README.md # Documentación general

|  |  |
| --- | --- |
| Archivo | Responsabilidad |
| game.py | Modelo del dominio, validación de movimientos, generador de estados |
| search.py | Implementación de BFS, DFS, A\*, IDA\* con métricas |
| heuristics.py | Factory de funciones heurísticas admisibles |
| water\_sort\_solver.py | Punto de entrada CLI con argparse |
| water\_sort\_gui.py | Aplicación Tkinter con visualización y análisis |

# Arquitectura de la aplicación

## 1. Representación de Estados

Los estados se representan como tuplas inmutables anidadas para permitir hashing:

State = Tuple[Tuple[str, ...], ...]

# Ejemplo de estado con 3 tubos

state: State = (

    ('R', 'G', 'B', 'Y'),  # Tubo 0: lleno mezclado

    ('R', 'R'),            # Tubo 1: parcialmente lleno

    ()                     # Tubo 2: vacío

)

**Ventajas de esta representación:**

* Inmutabilidad garantiza integridad en estructuras de datos
* Hashable → compatible con sets y diccionarios
* Comparaciones eficientes por identidad

## 2. Clase WaterSortGame

Encapsula las reglas del juego y la generación de estados:

class WaterSortGame:

    def \_\_init\_\_(self, num\_tubes: int, num\_colors: int,

                 capacity: int = 4, seed: int | None = None):

        """

        Args:

            num\_tubes: Cantidad de tubos (5-12)

            num\_colors: Cantidad de colores (3 a num\_tubes-2)

            capacity: Unidades por tubo completo (default 4)

            seed: Semilla para reproducibilidad

        """

**Métodos Clave**

**Generación de Puzzles:**

def generate\_initial\_state(self, scramble\_moves: int = 60) -> State:

    """Genera configuración aleatoria solucionable."""

**Validación de Estados:**

def is\_goal\_state(self, state: State) -> bool:

    """Verifica si todos los tubos no vacíos están completos y homogéneos."""

**Generación de Movimientos:**

def get\_valid\_moves(self, state: State) -> List[Move]:

    """

    Retorna movimientos legales desde un estado.

    Reglas:

    - Solo verter desde tubo no vacío

    - Solo a tubo con espacio

    - El color superior destino debe coincidir (si no está vacío)

    """

**Aplicación de Movimientos:**

def apply\_move(self, state: State, move: Move) -> Tuple[State, int]:

    """

    Aplica un movimiento y retorna (nuevo\_estado, unidades\_vertidas).

    Vierte toda la secuencia contigua del color superior.

    """

## 3. Clase SearchAlgorithms

Implementa los cuatro algoritmos con instrumentación de métricas:

@dataclass

class SearchResult:

    success: bool

    moves: List[Move]

    explored\_nodes: int      # Nodos visitados

    expanded\_nodes: int      # Nodos expandidos

    max\_frontier\_size: int   # Tamaño máximo de frontera

    depth: int               # Profundidad de la solución

    time\_seconds: float      # Tiempo de ejecución

# Heurísticas Disponibles

Todas las heurísticas son **admisibles** (nunca sobreestiman el coste real) y **consistentes**.

## 4.1 Entropy (Entropía) - DEFAULT

**Concepto:** Penaliza la dispersión de colores a través de múltiples tubos.

def build\_entropy\_heuristic(game: WaterSortGame) -> Heuristic:

    def heuristic(state: State) -> int:

        color\_distribution: Dict[str, list[int]] = defaultdict(list)

        for tube in state:

            counts = Counter(tube)

            for color, amount in counts.items():

                color\_distribution[color].append(amount)

        penalty = 0

        for color, amounts in color\_distribution.items():

            if len(amounts) <= 1:

                continue

            total\_units = sum(amounts)

            max\_in\_single\_tube = max(amounts)

            units\_outside = total\_units - max\_in\_single\_tube

            penalty += (len(amounts) - 1) \* units\_outside

        return penalty

    return heuristic

**Ventajas:**

* Favorece consolidación de colores
* Buen rendimiento en puzzles complejos

## 4.2 Completion (Completitud)

**Concepto:** Recompensa tubos completos y penaliza tubos incompletos.

def build\_completion\_heuristic(game: WaterSortGame) -> Heuristic:

    def heuristic(state: State) -> int:

        tubes\_incomplete = 0

        colors\_positioned = 0

        for tube in state:

            if not tube:

                continue

            if len(tube) == game.capacity and len(set(tube)) == 1:

                continue  # Tubo completo, ignorar

            tubes\_incomplete += 1

            base\_color = tube[0]

            streak = 1

            for color in tube[1:]:

                if color == base\_color:

                    streak += 1

                else:

                    break

            colors\_positioned += streak

        return (tubes\_incomplete \* game.capacity) - colors\_positioned

    return heuristic

**Ventajas:**

* Incentiva completar tubos rápidamente
* Eficaz en estados cercanos a la solución

## 4.3 Blocking (Bloqueo)

**Concepto:** Penaliza tubos mezclados y unidades bloqueadas por otros colores.

def build\_blocking\_heuristic(game: WaterSortGame) -> Heuristic:

    def heuristic(state: State) -> int:

        total\_mixed = 0

        blocked\_units = 0

        for tube in state:

            if not tube:

                continue

            # Penalizar mezcla

            if len(set(tube)) > 1:

                total\_mixed += len(tube)

            # Contar unidades bloqueadas

            for idx, color in enumerate(tube[:-1]):

                above = tube[idx + 1:]

                if any(upper != color for upper in above):

                    blocked\_units += 1

        return total\_mixed + (2 \* blocked\_units)

    return heuristic

**Ventajas:**

* Detecta configuraciones difíciles de desbloquear
* Útil para evitar dead-ends

# 5. Análisis comparativo

Para llevar a cabo este análisis vamos a usar:

* 6 tubos
* 4 colores
* Número semilla 10

## Resultados

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmo | Heurístico | Tiempo de ejecución | Nodos explorados | Movimientos |
| BFS | N/A | 0.080 | 7179 | 11 |
| DFS | N/A | 0.004 | 324 | 323 |
| A-STAR | ENTROPY | 0.005 | 93 | 11 |
| A-STAR | COMPLETION | 0.002 | 57 | 11 |
| A-STAR | BLOCKING | 0.001 | 19 | 12 |
| IDA | ENTROPY | 0.001 | 36 | 13 |
| IDA | COMPLETION | 0.005 | 163 | 13 |
| IDA | BLOCKING | 0.001 | 55 | 38 |

A screen shot of a computer

AI-generated content may be incorrect.

Para mejorar la visualización hemos eliminado el algoritmo BFS ya que su tiempo de ejecución es mucho mayor que el de los otros algoritmos.

A screen shot of a graph

AI-generated content may be incorrect.

**Observaciones:**

**A\*-Completion** es el que da la mejor solución en el menor tiempo mientras que el algoritmo más rápido fue **A\*-Blocking**, que encontró la solución solo explorando **19 nodos**, pero su solución es algo peor (1 movimiento más).

**IDA-Completion** y **A\*-Entropy** han logrado resultados decepcionantes, explorando demasiados nodos a comparación de otros heurísticos.

**BFS es el menos eficiente**, explorando 7179 nodos mientras que **DFS da una solución pésima de 324 movimientos.** Lo cuál era predecible tratándose de algoritmos no informados.

# Casos de prueba

Para los casos de prueba hemos diseñado un script que compara 8 casos:

1. Configuración mínima (5 tubos y 3 colores)

A screenshot of a computer

AI-generated content may be incorrect.

El más rápido fue IDA-completion el que exploró menos nodos A\*-entropy que además encontró la solución de menor coste

1. Configuración estándar (8 tubos, 6 colores)

A screenshot of a computer screen

AI-generated content may be incorrect.

El más rápido fue IDA-Blocking pero el que exploró menos nodos fue A\*blocking que encontró una solución muchomejor. La mejor solución la ha encontrado BFS.

1. Escalabilidad progresiva A\*-blocking(mismo algoritmo complejidad creciente)

A screenshot of a computer

AI-generated content may be incorrect.

Vemos que el incremento es más o menos progresivo excepto en el paso de 6 a 7 tubos, que encuentra un amejor solución para 7 tubos que para 6.

1. Escalabilidad progresiva IDA-entropy(mismo algoritmo complejidad creciente)

A screenshot of a computer

AI-generated content may be incorrect.

Es progresiva con grandes saltos, funciona peor que A\*.

1. Comparación de heurísticas (9 tubos 7 colores)

A screenshot of a computer

AI-generated content may be incorrect.

En este caso el que mejor ha funcionado ha sido A\*-blocking consiguiendo la mejor solución de todas las encontradas en el menor número de nodos explorados posibles y tiempo.

1. Comparación de heurísticas (11 tubos 9 colores)

A screenshot of a computer screen

AI-generated content may be incorrect.

Una vez más A\*-blocking vuelve a ganar. Esta vez por goleada. IDA-completion estaba tardando una eternidad así que la cancelé.

1. Robustez con Distintas Semillas A\*-blocking (7 tubos, 5 colores)

A screenshot of a computer screen

AI-generated content may be incorrect.

Al cambiar la semilla sigue consiguiendo buenísimos resultados, es robousto.

1. Robustez con Distintas Semillas para IDA\*

A screenshot of a computer screen

AI-generated content may be incorrect.

En este caso se sigue consiguiendo soluciones más dispares que en A\*.

1. Casos Extremos

A screenshot of a computer

AI-generated content may be incorrect.

A\* gana en todas las categorías a IDA, en dos de esos casos IDA ni se completó.

# Conclusión

El algoritmo que mejor funciona es **A\*, en casi todas las pruebas realizadas**, consiguiendo soluciones mejores en menor tiempo que IDA\*.

Por último, **el mejor de todos los heurísticos para A\* es sin duda blocking**. Ha demostrado dar las soluciones en **menor tiempo** que cualquier otro. Es cierto que en problemas más simples entropy y completion han dado soluciones ligeramente mejores (1-2 movimientos de diferencia), pero cuando pasamos a problemas complejos, es cierto que los tres daban la misma solución, pero A\*-blocking siempre lo hacía explorando muchísimos menos nodos y en menortiempo