Water Sort Puzzle

Eduardo González Bautista
Juan manuel Valenzuela González

ÍNDICE

1. 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.

2. 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

3.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
```

4.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

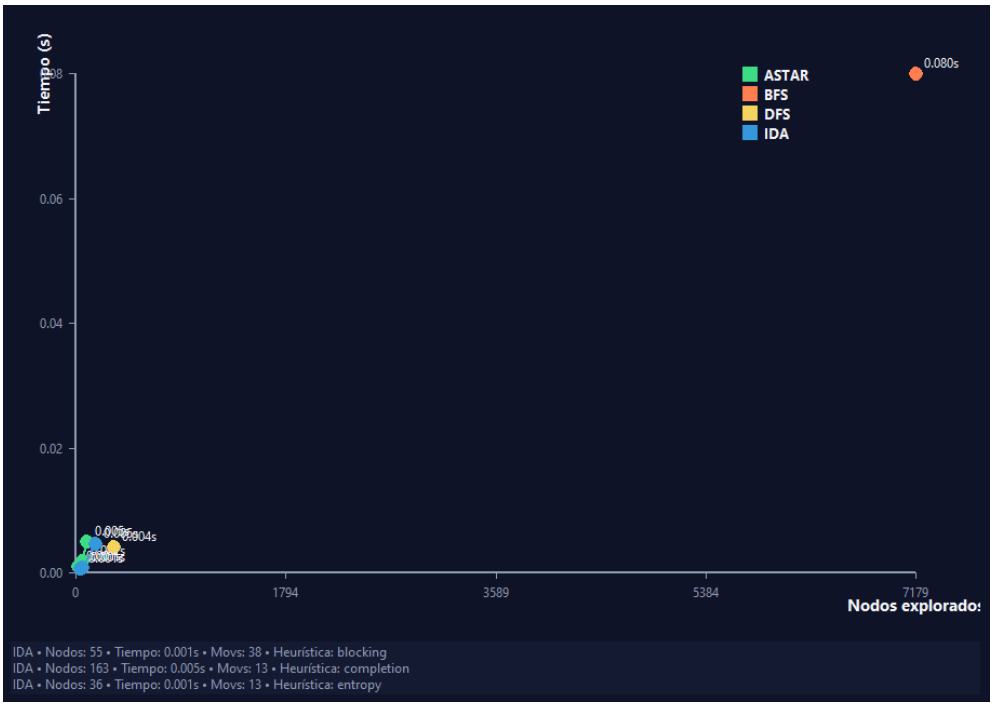
5. Análisis comparativo

Para llevar a cabo este análisis vamos a usar:

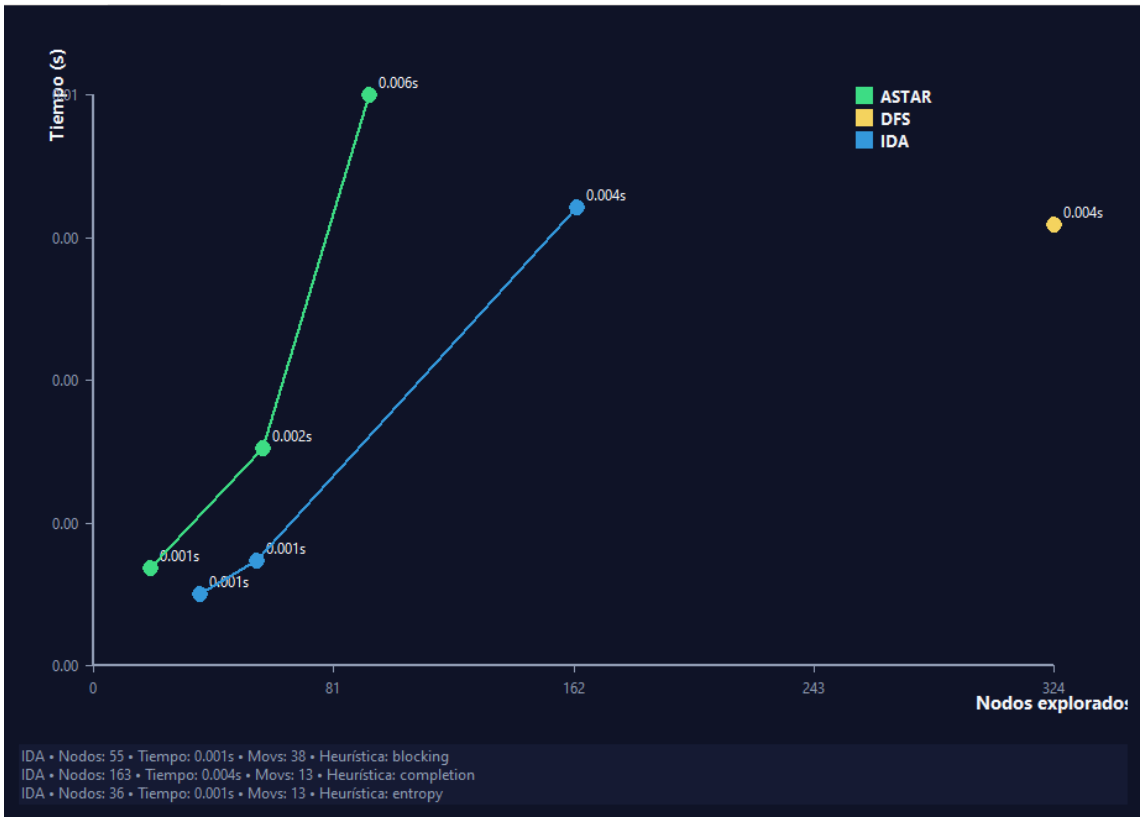
- 6 tubos
- 4 colores
- Número semilla 10

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



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



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.

6.Casos de prueba

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

1. Configuración mínima (5 tubos y 3 colores)

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
BFS	N/A	5	3	42	✅	7	780	480	303	0.0079
DFS	N/A	5	3	42	✅	65	66	65	66	0.0007
ASTAR	entropy	5	3	42	✅	7	8	7	32	0.0006
ASTAR	completion	5	3	42	✅	7	20	19	54	0.0006
ASTAR	blocking	5	3	42	✅	8	14	13	45	0.0005
IDA	entropy	5	3	42	✅	11	34	12	12	0.0005
IDA	completion	5	3	42	✅	9	45	21	10	0.0004
IDA	blocking	5	3	42	✅	28	35	28	29	0.0005

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

2. Configuración estándar (8 tubos, 6 colores)

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
BFS	N/A	8	6	100	✅	20	293389	184572	108845	4.1106
DFS	N/A	8	6	100	✅	100	1172	482	101	0.0148
ASTAR	entropy	8	6	100	✅	21	44	43	190	0.0048
ASTAR	completion	8	6	100	✅	21	1618	1617	3756	0.0512
ASTAR	blocking	8	6	100	✅	21	26	25	94	0.002
IDA	entropy	8	6	100	✅	40	296	78	41	0.0054
IDA	completion	8	6	100	✅	25	3041	38013	26	1.4613
IDA	blocking	8	6	100	✅	60	80	60	61	0.0018

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.

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

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
ASTAR	blocking	5	3	500	✅	6	11	10	33	0.0004
ASTAR	blocking	6	4	500	✅	14	20	19	60	0.0008
ASTAR	blocking	7	5	500	✅	13	20	19	78	0.0011
ASTAR	blocking	8	6	500	✅	19	27	26	101	0.0019
ASTAR	blocking	9	7	500	✅	22	28	27	107	0.0024
ASTAR	blocking	10	8	500	✅	24	39	38	180	0.0045
ASTAR	blocking	11	9	500	✅	29	54	53	153	0.005
ASTAR	blocking	12	10	500	✅	33	74	73	198	0.0066

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.

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

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
IDA	entropy	5	3	500	✅	8	19	9	9	0.0004
IDA	entropy	7	5	500	✅	21	79	26	22	0.0015
IDA	entropy	9	7	500	✅	39	104	51	40	0.0026
IDA	entropy	11	9	500	✅	57	101	127	58	0.0054

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

5. Comparación de heurísticas (9 tubos 7 colores)

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
ASTAR	entropy	9	7	999	✅	24	44	43	196	0.0048
ASTAR	completion	9	7	999	✅	24	1756	1755	2568	0.0523
ASTAR	blocking	9	7	999	✅	24	34	33	134	0.0023
IDA	entropy	9	7	999	✅	59	88	60	60	0.0027
IDA	completion	9	7	999	✅	29	3336	520298	30	18.3491
IDA	blocking	9	7	999	✅	70	98	72	72	0.0028

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.

6. Comparación de heurísticas (11 tubos 9 colores)

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
ASTAR	entropy	11	9	777	✅	31	2588	2587	11628	0.3122
ASTAR	completion	11	9	777	✅	31	8806	8805	28230	0.4407
ASTAR	blocking	11	9	777	✅	31	42	41	179	0.004
IDA	entropy	11	9	777	✅	57	226	74	58	0.0059
IDA	entropy	11	9	777	✅	57	226	74	58	0.0077
IDA	completion	11	9	777	❌	—	0	0	0	0
IDA	blocking	11	9	777	✅	88	112	94	89	0.0031

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

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

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
ASTAR	blocking	7	5	42	✅	13	19	18	67	0.001
ASTAR	blocking	7	5	123	✅	14	20	19	64	0.0012
ASTAR	blocking	7	5	256	✅	15	22	21	80	0.0014
ASTAR	blocking	7	5	789	✅	17	24	23	77	0.0023
ASTAR	blocking	7	5	999	✅	15	25	24	99	0.0018

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

8. Robustez con Distintas Semillas para IDA*

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
IDA	entropy	8	6	42	✅	35	106	42	36	0.0024
IDA	entropy	8	6	100	✅	40	296	78	41	0.0063
IDA	entropy	8	6	333	✅	35	78	37	36	0.0015
IDA	entropy	8	6	666	✅	37	141	47	38	0.003
IDA	entropy	8	6	999	✅	30	441	113	31	0.0142

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

9. Casos Extremos

Algoritmo	Heurística	Tubos	Colores	Semilla	Éxito	Movs	Nodos Expl.	Nodos Exp.	Frontera	Tiempo (s)
ASTAR	blocking	12	10	1234	✅	34	103	102	193	0.0068
IDA	blocking	12	10	1234	❌	—	0	0	0	0
ASTAR	entropy	6	4	321	✅	14	75	74	230	0.0046
IDA	entropy	6	4	321	✅	21	79	32	22	0.0017
ASTAR	blocking	10	3	555	✅	11	399	398	3821	0.0522
IDA	blocking	10	3	555	❌	—	0	0	0	0

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

7.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