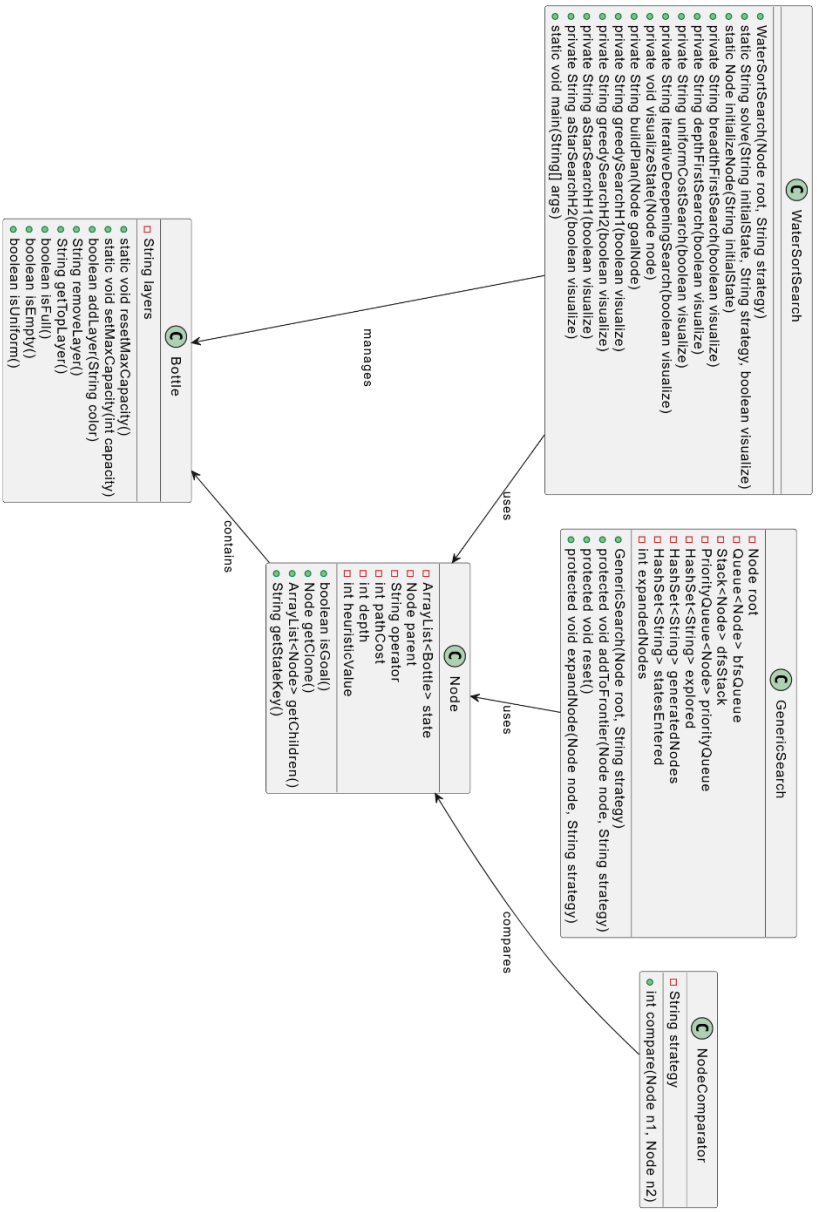
# AI Project Report

## A-Class diagrams:




## B-Search strategies:

### 1-BFS:

**Queue-based Exploration**: The BFS algorithm uses a **FIFO (First-In-First-Out) queue** to explore nodes level by level. (General description before I dive into details)

- The algorithm processes nodes starting from the root node, exploring each child before moving to the next sibling.
- I firstly initialize the queue with the starting node(root)
- Then, node expansion, In a while loop, the algorithm dequeues a node, checks if it's the goal, and if not, expands its children and enqueues them for future exploration.
- when the goal state is found, the search stops and the path from the root is returned.

# 2-DFS:

**Stack-based Exploration**: The DFS algorithm uses a **LIFO (Last-In-First-Out) stack** to explore nodes by diving as deep as possible along each branch before backtracking. (General description before I dive into details)

Note: We are leveraging a stack to expand the deepest node first.

- Similar to BFS, a stack is initialized with the starting node
- The algorithm pops a node from the stack, checks if it's the goal, and if not, expands its children. Instead of enqueuing, the children are pushed onto the stack, so they are explored later in reverse order
- when the goal state is found, the search stops and the path from the root is returned.

# 3-UCS:

**Priority Queue Exploration**: UCS uses a priority queue to expand the node with the lowest cumulative cost first. using a priority queue to ensure the node with the minimum path cost is processed next.

- The priority queue holds nodes sorted by their path cost ( the accumulated cost from the root node).
- The algorithm dequeues the node with the lowest cost, checks if it's the goal, and expands it. Each child node's cost is calculated by adding the cost of the current node to the cost of the action leading to the child.
- The NodeComparator class helps compare nodes based on their total cost so I can use it easily and freely in the implementation of strategy itself.
- when the goal state is found, the search stops and the path from the root is returned.

# 4-ID:

The search starts with a depth limit of 0. At each iteration, the depth limit is incremented, allowing for deeper searches. (General description before I dive into details)

**Stack Initialization**: The search uses a stack (dfsStack) to perform DFS. Initially, the root node is pushed onto the stack.

- **Main Loop**: The main loop continues until the stack is empty or a solution is found. In each iteration, a node is popped from the stack, and the following checks are performed:

1. **Goal Check**: If the node is the goal, the plan is returned along with the path cost and the number of expanded nodes.
2. **Depth Limit Check**: If the current node's depth matches the current depth limit, and the stack is empty (i.e., all nodes at the current depth have been explored), the algorithm increments the depth limit and reinitializes the stack with a clone of the root node., and I clear the explored and generated nodes.

- If the current node's depth is less than the depth limit, it is expanded. The expandNode method is called to generate and explore child nodes.
- If the stack is empty and certain states have not been entered, the algorithm reinitializes the stack with a clone of the root node and increments the depth limit.
- when the goal state is found, the search stops and the path from the root is returned.

# Greedy and A*:

Implementation of Greedy and A Search Algorithms*

For Greedy Search and A* Search, I used two distinct heuristic functions, distinguishing between GR1/AS1 and GR2/AS2. The algorithms are managed by expanding nodes from a priority queue, guided by their heuristic values and path costs.

**Node Class:**

In the Node class, the heuristic value is initialized to 0 when the node is created. Later, the calculateHeuristics(String strategy) method calls either calculateH1() or calculateH2() based on the strategy.

calculateH1() computes the number of non-uniform bottles.

calculateH2() computes the number of misplaced layers across all bottles.

NodeComparator Class:

The NodeComparator class compares nodes in the priority queue based on their strategy:

The compare(Node n1, Node n2) method uses h(n) for Greedy and f(n) = g(n) + h(n) for A*.

**Greedy Search:**

Greedy search expands nodes based on their heuristic value alone:

greedySearchH1() (for GR1) uses calculateH1() to prioritize states with fewer non-uniform bottles.

greedySearchH2() (for GR2) uses calculateH2() to focus on states with fewer misplaced layers.

**A Search*:**

A* combines path cost (g(n)) and heuristic value (h(n)) to expand nodes:

aStarSearchH1() (for AS1) uses calculateH1() for the number of non-uniform bottles.

aStarSearchH2() (for AS2) uses calculateH2() for misplaced layers.

Both algorithms use the priority queue to ensure that nodes with the smallest heuristic or total cost (f(n)) are expanded first.

**Heuristic Functions and Their Admissibility**

Heuristic 1 (H1): Number of Non-Uniform Bottles

Function: Counts how many bottles still contain mixed colors.

Use: Guides GR1 and AS1 by prioritizing states with more uniform bottles.

Admissibility: H1 is admissible because it never overestimates the number of steps required to achieve full bottle uniformity, providing a lower bound.

Heuristic 2 (H2): Number of Misplaced Layers

Function: Counts the total number of misplaced layers in all bottles.

Use: Guides GR2 and AS2 by focusing on states with fewer misplaced layers.

Admissibility: H2 is admissible since it does not overestimate the actions needed to correctly place all layers, making it a safe estimate.

# Simple Generic Comparison:

| Test | Algorithm | Runtime (ms) | Nodes Expanded | Cost | CPU Utilization (%) | Optimality | Completeness | RAM Usage |
|---|---|---|---|---|---|---|---|---|
| **Test 0** | BFS | 91 | 15 | 10 | 2.74% | Optimal | Complete | Low |
| | DFS | 3 | 9 | 18 | 0.09% | Not Optimal | Complete | Low |
| | UCS | 3 | 15 | 10 | 0.09% | Optimal | Complete | Low |
| | ID | 21 | 58 | 14 | 0.63% | Optimal | Complete | Low |
| | Greedy 1 | 3 | 11 | 10 | 0.09% | Not Optimal | Complete | Low |
| | Greedy 2 | 2 | 9 | 10 | 0.06% | Not Optimal | Complete | Low |
| | A* 1 | 2 | 15 | 10 | 0.06% | Optimal | Complete | Low |
| | A* 2 | 2 | 13 | 10 | 0.06% | Optimal | Complete | Low |
| **Test 1** | BFS | 481 | 1269 | 10 | 14.5% | Optimal | Complete | Medium |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | DFS | 14 | 53 | 132 | 0.42% | Not Optimal | Complete | Low |
| | UCS | 147 | 883 | 11 | 4.43% | Optimal | Complete | Medium |
| | ID | 222 | 1422 | 16 | 6.68% | Optimal | Complete | Medium |
| | Greedy 1 | 7 | 22 | 13 | 0.21% | Not Optimal | Complete | Low |
| | Greedy 2 | 10 | 42 | 19 | 0.30% | Not Optimal | Complete | Low |
| | A* 1 | 118 | 677 | 11 | 3.56% | Optimal | Complete | Medium |
| | A* 2 | 46 | 216 | 11 | 1.38% | Optimal | Complete | Medium |
| **Test 2** | BFS | 430 | 1269 | 10 | 12.94% | Optimal | Complete | Medium |
| | DFS | 12 | 53 | 132 | 0.36% | Not Optimal | Complete | Low |
| | UCS | 148 | 883 | 11 | 4.45% | Optimal | Complete | Medium |
| | ID | 207 | 1422 | 16 | 6.24% | Optimal | Complete | Medium |
| | Greedy 1 | 7 | 22 | 13 | 0.21% | Not Optimal | Complete | Low |
| | Greedy 2 | 11 | 42 | 19 | 0.33% | Not Optimal | Complete | Low |
| | A* 1 | 122 | 677 | 11 | 3.68% | Optimal | Complete | Medium |
| | A* 2 | 48 | 216 | 11 | 1.45% | Optimal | Complete | Medium |
| **Test 3** | BFS | 19 | 78 | 20 | 0.57% | Not Optimal | Complete | Low |
| | DFS | 4 | 18 | 23 | 0.12% | Not Optimal | Complete | Low |
| | UCS | 8 | 70 | 20 | 0.24% | Not Optimal | Complete | Low |
| | ID | 32 | 337 | 23 | 0.96% | Optimal | Complete | Medium |
| | Greedy 1 | 3 | 13 | 20 | 0.09% | Not Optimal | Complete | Low |
| | Greedy 2 | 3 | 14 | 20 | 0.09% | Not Optimal | Complete | Low |
| | A* 1 | 8 | 62 | 20 | 0.24% | Optimal | Complete | Low |
| | A* 2 | 11 | 48 | 20 | 0.33% | Optimal | Complete | Low |
| **Test 4** | BFS | 441 | 1798 | 10 | 13.31% | Optimal | Complete | Medium |
| | DFS | 20 | 77 | 157 | 0.60% | Not Optimal | Complete | Low |
| | UCS | 221 | 1194 | 10 | 6.67% | Optimal | Complete | Medium |
| | ID | 173 | 1147 | 11 | 5.21% | Optimal | Complete | Medium |
| | Greedy 1 | 4 | 10 | 11 | 0.12% | Optimal | Complete | Low |
| | Greedy 2 | 3 | 10 | 11 | 0.09% | Optimal | Complete | Low |
| | A* 1 | 130 | 732 | 10 | 3.91% | Optimal | Complete | Medium |
| | A* 2 | 83 | 453 | 10 | 2.50% | Optimal | Complete | Medium |
| | | | | | | | | |

| **Test (RAM Usage)** | **BFS** | **DFS** | **UCS** | **IDS** | **Greedy H1** | **Greedy H2** | ***A H1**** | ***A H2**** |
|---|---|---|---|---|---|---|---|---|
| **Test 0** | **463,688** | **0** | **0** | **0** | **1,036,696** | **1,036,696** | **0** | **0** |
| **Test 1** | **12,761,912** | **3,143,680** | **48,234,496** | **-81,398,144** | **1,034,648** | **2,097,200** | **38,785,440** | **12,582,912** |
| **Test 2** | **69,206,016** | **3,143,680** | **-108,295,056** | **74,437,016** | **2,071,352** | **3,133,848** | **37,748,736** | **12,582,912** |
| **Test 3** | **2,097,152** | **0** | **3,133,848** | **12,582,960** | **1,036,664** | **0** | **3,133,848** | **2,097,152** |
| **Test 4** | **-40,152,704** | **5,244,928** | **75,497,448** | **-80,858,048** | **0** | **1,036,696** | **-106,097,072** | **30,406,656** |

# Comments on RAM Usage:

1. **BFS:**
    - **Typically consumes substantial memory. The largest memory usage occurred in Test 2, where it used over 69 MB. In Test 4, however, a negative memory value (-40 MB) was recorded, suggesting memory release or reallocation.**
    - **Overall, BFS's memory usage increases significantly as the problem space grows, which aligns with its known space complexity of O(b^d), where b is the branching factor and d is the depth of the shallowest solution.**
2. **DFS:**

- **DFS shows extremely low memory usage across all tests, with most results being 0 bytes. In Tests 1 and 2, it used around 3 MB, which is minimal compared to other strategies. This is expected since DFS only needs to store a single path from the root to a leaf node, making it space-efficient (O(bd)).**

3. **UCS:**
    - **Shows very high memory usage in some tests, such as Test 1 (48 MB) and Test 4 (75 MB). However, UCS also records extremely negative memory usage in Tests 2 and 4, indicating potential memory release or deallocation during the test.**
    - **UCS tends to require a lot of memory due to maintaining priority queues, leading to high space complexity, especially when the search space is large.**
4. **IDS:**
    - **Similar to UCS, IDS has cases of memory release (Test 1: -81 MB, Test 4: -80 MB). This could be due to memory reallocation during iterative deepening.**
    - **IDS's memory usage is generally lower than BFS, but still fluctuates, showing both high positive values (74 MB in Test 2) and negative values.**
5. **Greedy H1 & H2:**
    - **Both Greedy H1 and H2 have consistently low memory usage across all tests, with no major spikes. The highest usage is around 3 MB (Test 2).**
    - **These algorithms are memory-efficient since they focus on exploring the most promising nodes based on heuristic functions, but they do not guarantee optimal solutions.**
6. ***A H1 & H2*:**
    - **A* H1 and H2 tend to use moderate memory, with higher memory usage in Test 1 (A* H1 at 38 MB). Like UCS, A* also uses priority queues to store nodes, which can lead to higher memory consumption.**
    - **In Test 4, A* H1 shows a negative value (-106 MB), indicating possible memory release or reallocation.**
    - **A* strikes a balance between memory usage and completeness, as it aims to find the optimal solution using both path cost and heuristics.**

# Other Comments Section:

**1. Breadth-First Search (BFS)**

- **CPU Utilization**: BFS shows higher CPU utilization in most tests (e.g., Test 0: 2.74%, Test 1: 14.5%). BFS systematically explores all nodes level by level, leading to a large number of nodes being expanded and hence higher CPU consumption.

- **Runtime**: BFS is generally slower because it explores many nodes, particularly in Test 4 where it takes 441 ms. The time increases because BFS doesn't employ any heuristic, so it ends up examining all possible nodes to find the solution.
- **Nodes Expanded**: BFS is guaranteed to explore all nodes at each level, leading to large numbers of expanded nodes (e.g., 1269 nodes in Test 1, 1798 nodes in Test 4).
- **Cost**: BFS maintains a low cost, which is expected since it explores the nodes without any cost-based prioritization. This shows BFS typically finds the solution with minimal cost but can be inefficient in terms of runtime.
- **Optimality and Completeness**: BFS is both **optimal** and **complete** because it explores all nodes level by level and guarantees the shortest path (least-cost path) in an unweighted graph or when all moves have the same cost.

**2. Depth-First Search (DFS)**

- **CPU Utilization**: DFS has the lowest CPU utilization in most tests (e.g., Test 0: 0.09%, Test 1: 0.42%). This is because DFS explores nodes in a deep manner, expanding only a single path at a time.
- **Runtime**: DFS is faster in most cases, but it is not optimal. For example, it completes Test 3 in 4 ms and Test 0 in 3 ms.
- **Nodes Expanded**: DFS expands fewer nodes in comparison to BFS, but this can lead to longer search times in larger graphs or suboptimal paths. This is reflected in **Test 4**, where DFS expanded just 77 nodes but took 20 ms.
- **Cost**: In many cases, DFS incurs a high cost (e.g., Test 4: 157), which shows that it doesn’t always find the optimal solution. The cost increases because DFS can go down paths that are suboptimal.
- **Optimality**: DFS is not optimal because it can reach a solution through a long, non-optimal path (not the least cost path).
- **Completeness**: DFS is complete in finite spaces but might fail in infinite or cyclic graphs if not handled properly with backtracking.

**3. Uniform Cost Search (UCS)**

- **CPU Utilization**: UCS shows moderate CPU utilization across the tests (e.g., Test 1: 4.43%, Test 2: 4.45%). UCS prioritizes nodes based on the cost to reach them, which increases the number of nodes expanded but not as much as BFS.
- **Runtime**: UCS takes longer in comparison to DFS but is faster than BFS. For instance, UCS runs in 147 ms in Test 1 and 148 ms in Test 2.
- **Nodes Expanded**: UCS expands more nodes than DFS but fewer than BFS. The expanded nodes are prioritized based on cost, and thus the number of expanded nodes is influenced by the structure of the graph.
- **Cost**: UCS shows consistent costs across the tests (e.g., Test 1: 11), reflecting its behavior as an optimal search strategy based on path cost.
- **Optimality**: UCS is **optimal** because it ensures that it finds the path with the lowest cost.
- **Completeness**: UCS is **complete**, as it will find a solution if one exists, assuming all edge costs are positive.

**4. Iterative Deepening (ID)**

- **CPU Utilization**: ID has moderate CPU utilization (e.g., Test 1: 6.68%). ID is essentially a combination of DFS and BFS, running DFS repeatedly with increasing depth limits. This causes it to expand nodes more than DFS but fewer than BFS.
- **Runtime**: ID performs slower than DFS but faster than BFS and UCS in most cases. For example, Test 1 shows 222 ms, while Test 4 shows 173 ms.
- **Nodes Expanded**: ID expands more nodes than DFS, as it repeats the search multiple times, but not as many as BFS. The number of nodes expanded in Test 1 (1422) reflects the repeated searches over increasing depths.
- **Cost**: ID's cost is higher than DFS but lower than BFS and UCS in most cases. In Test 1, the cost is 16, reflecting its balanced nature of performing depth-limited searches.
- **Optimality**: ID is **optimal** for graphs with finite branching, as it eventually explores all nodes at a given depth level.
- **Completeness**: ID is **complete** and can solve any problem with a finite search space, just like BFS.

**5. Greedy Best-First Search (GR1 and GR2)**

- **CPU Utilization**: Greedy algorithms show very low CPU utilization (e.g., Test 1: 0.21% for GR1). Since they prioritize nodes based on a heuristic without considering cost, they expand very few nodes quickly.
- **Runtime**: Greedy algorithms run very fast in all tests. For example, Test 1 GR1 runs in just 7 ms, and Test 3 GR1 runs in 3 ms.
- **Nodes Expanded**: Greedy algorithms expand significantly fewer nodes (Test 1: 22 nodes for GR1) since they only focus on the most promising node according to the heuristic.
- **Cost**: The cost is high in most cases, reflecting their lack of consideration for actual path costs. For example, GR1 cost is 13 in Test 1, which is suboptimal.
- **Optimality**: Greedy algorithms are **not optimal**, as they only prioritize nodes based on heuristic estimates, potentially leading to suboptimal solutions.
- **Completeness**: Greedy algorithms are not guaranteed to find a solution if one exists, especially if the heuristic misguides the search.

**6. *A Search (A*1 and A*2)****

- **CPU Utilization**: A* has moderate CPU utilization (e.g., Test 1: 3.56% for A*1). A* balances the path cost and heuristic, so it expands fewer nodes than BFS but more than Greedy algorithms.
- **Runtime**: A* has moderate runtime performance. For instance, Test 1 shows A*1 with 118 ms and A*2 with 46 ms. It performs better than BFS and UCS in terms of nodes expanded but can still be slower than Greedy.
- **Nodes Expanded**: A* expands a moderate number of nodes (e.g., 677 in Test 1 for A*1), benefiting from the balance of path cost and heuristic guidance.
- **Cost**: A* shows consistent cost values (e.g., 11 in most tests), confirming its optimal path-finding abilities.
- **Optimality**: A* is **optimal** when the heuristic is admissible (i.e., it never overestimates the true cost).
- **Completeness**: A* is **complete**, given that the heuristic is admissible and the graph is finite.