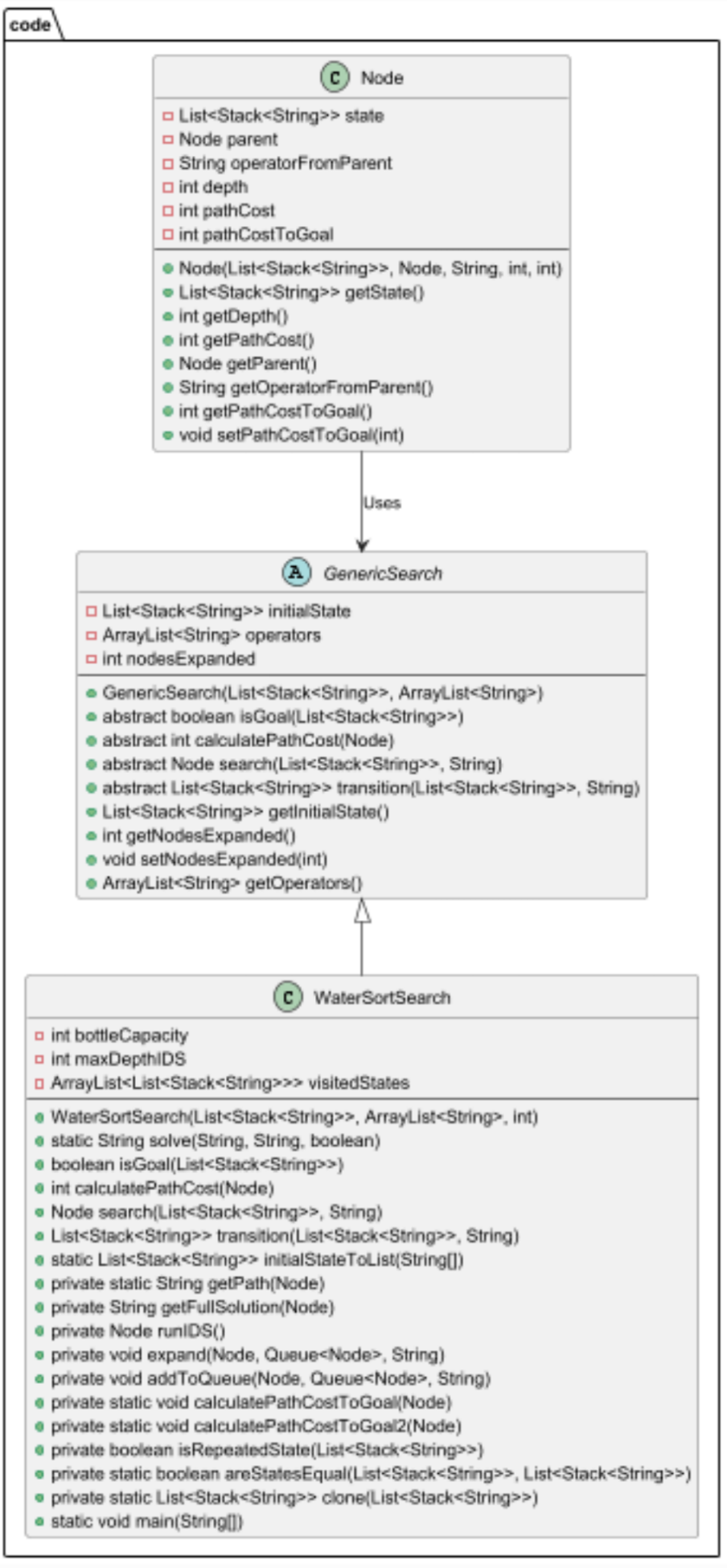
# [Class Diagram]



[Heuristic Functions]

We used 2 admissible heuristic functions:

1) Number of bottles that have more than one color
This heuristic was used in GR1 and AS1.
This heuristic counts how many bottles contain layers of more than one color.
As the number increases this indicates that this node is farther from the goal, as more bottles need to be edited.
This heuristic is admissible because it never overestimates the cost to the goal, because to make a faulty bottle into a bottle with only one color or empty you need at least to pour one layer at best case.

2) Total Number of Layers That Are Not of the Same Color as the Bottom Layer
This heuristic was used in GR2 and AS2.
This heuristic counts the total number of layers across all bottles that are not the same color as the bottom layer of their respective bottles.
As the number increases this indicates that this node is farther from the goal, as more layers are misplaced.
We chose the bottom layer because usually it is the color that the bottle ends up with.
This heuristic is admissible because it never overestimates the cost to the goal, because for a misplaced layer to be placed correctly it needs at least one action, also the layers of the same color of the bottom layer that will need to be poured to get to a layer under are neglected, and this makes

the actual cost higher.

[Implementation]

We represented a state as the bottles and the layers inside them, as a list of stacks (the bottles) of strings (the layers).

We formulate the problem in the solve method then call the search method, where each strategy is implemented in a specific way.

But for the ID there is a method runIDS between solve and search to run the IDS loop for trying different depths and each Iteration the search is called until a goal is found.

In the search we initialize the search queue, it is a priority queue if the strategy is UCS, Greedy or A*, otherwise it is a normal queue implemented using a linked list. We add to it the root node, and also we add the state of the root node to an arraylist to track visited states.

If the queue is empty we return no solution, otherwise we enter a loop, remove the front node from the queue, if it is a goal we extract from the solution and return it, if the strategy is ID and we reached the maximum depth we go to the next iteration skipping the expansion, else we expand the current node using the expand method and increment the number of expanded nodes.

In the expand method each time we get a new node we call

addToQueue method, if the state of the new node is already in the visited states list then we ignore it, else we add its state to the visited states list.

Adding to the search queue logic:

BFS: the node is added at the rear of the queue using the predefined add method.

DFS, IDS: the node is added at the front of the queue using the predefined addFirst method.

UCS: the node is added at the right place of the priority queue using the predefined add method, the least layers poured first.

Greedy, A*: first the estimated cost from the node to the goal is set according to the heuristic function, then the node is added at the right place of the priority queue using the predefined add method, in the Greedy least h(n) cost first, in the A* least (layers poured + h(n) cost) first.

[Strategies Performance Comparison]

We ran the 5 grids in the public tests on each strategy to compare them.

Sample: (Grid1)  (path cost ; nodes expanded ; RAM usage in bytes)

BFS: 8 ; 536 ; 4600

DFS: 112 ; 47 ; 4400

UCS: 8 ; 400 ; 92776

IDS: 9 ; 448 ; 4512

GR1: 12 ; 17 ; 91432

GR2: 8 ; 9 ; 90792

AS1: 8 ; 182 ; 92664

AS2: 8 ; 43 ; 92664

1) Completeness

BFS: Complete

DFS: Incomplete

UCS: Complete

IDS: Complete

Greedy: Incomplete in general

A*: Complete, because the g(n) or number of layers poured is positive

Note: all search strategies got a solution in the public test cases provided.

2) Optimality

BFS: not optimal here, because the path cost (number of layers poured) of the nodes at depth d is
not greater than the cost of the nodes at depth d + 1.
However, it got the solution with the least cost in all public tests.

DFS: not optimal, it got the solution with the largest cost in all public tests.

UCS: optimal, as the cost of a node is less than or
equal the cost of its successors, the child nodes must have more layers poured than their parent, it got the solution with the least cost in all public tests.

IDS: not optimal, 3 of the 5 tests did not get the solution with the smallest cost.

Greedy: not optimal in general, in the public tests GR1 got nonoptimal solutions 2 times, and GR2 got nonoptimal solution once.

A*: optimal, given that our 2 heuristics are admissible, and it got the solution with the least cost in all public tests.

3) Number of Expanded Nodes

A* is optimally efficient., no other optimal search strategy will expand less nodes than A*, the only optimal strategy here with A* is UCS and in all public tests A* expanded less nodes than UCS.

BFS, UCS, IDS have the biggest number of expanded nodes in the public tests, in general.

DFS and Greedy have less number of expanded nodes in the public tests, compared to BFS, UCS, IDS.

GR2 got less than or equal expanded nodes than GR1 in all public tests.

4) RAM Usage

A) Java Runtime class:
Java provides the Runtime class, which can be used to check memory usage at runtime.
We used ChatGPT to explain to us the methods that this class provides.
This is the code we used to calculate the RAM usage of a search strategy in a specific test case:

```
// Get the runtime instance
    Runtime runtime = Runtime.getRuntime();

    // Run garbage collection to minimize memory noise
    runtime.gc();
```

```
        // Get memory usage before the method call
        long memoryBefore = runtime.totalMemory() -
runtime.freeMemory();

        // Call the solve method we want to measure
        String s = "5;" +
                "4;" +
                "b,y,r,b;" +
                "b,y,r,r;" +
                "y,r,b,y;" +
                "e,e,e,e;" +
                "e,e,e,e;";

    System.out.println(solve(s, "AS2", false));

        // Run garbage collection again
        runtime.gc();

        // Get memory usage after the method call
        long memoryAfter = runtime.totalMemory() -
runtime.freeMemory();

        // Calculate the memory used by the method in bytes
        long memoryUsed = memoryAfter - memoryBefore;
```

B) Comparison

High RAM usage: UCS, Greedy, A*

Low RAM usage: BFS, DFS, IDS

Reason: In the UCS, Greedy, A* we used priority queue to implement the search queue, while in the BFS, DFS, IDS we used linked list to implement the search queue. And the priority queue uses more memory than the linked list.

5) CPU Utilization

BFS: Moderate to High, due to Large number of node expansions.

DFS: Low to Moderate, Lower CPU usage if the goal is deep in the tree, but can increase significantly with backtracking and deep irrelevant searches.

IDS: High, Repeated expansion of nodes, especially near the root, increases CPU consumption.

UCS: Moderate to High, Priority queue sorting increases CPU load, and high number of nodes expansion.

Greedy: Moderate, Heuristic-based expansion reduces node expansion, but still uses a priority queue, adding some CPU overhead.

A*: High, Combines path cost and heuristic in priority queue, leading to fewer nodes expanded but high computational cost per node.