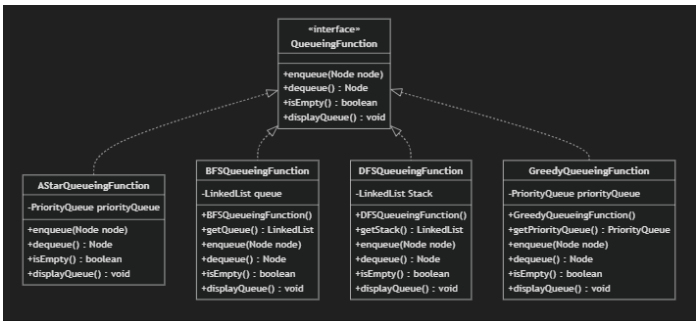
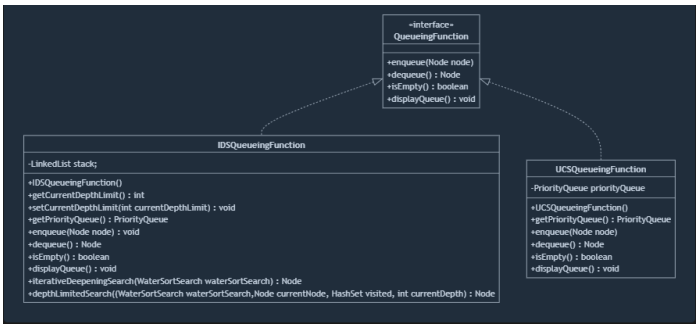
AI Sort Search Problem

Class diagram



To implement the search strategies, we use a general interface since most of the queueing functions have **enqueue**, **dequeue**, **isEmpty**, and **displayQueue** methods. We define the interface, and all the classes that implement this interface must implement these methods. For every method, we provide the corresponding implementation in each class.



For the **IDSQueueingFunction**, we added two methods because its logic involves a loop where each time it searches to a max depth, and in every iteration, the max depth increases by one. The **iterativeDeepeningSearch** method initializes the data structures and calls **depthLimitedSearch** to perform the search up to a certain depth.

For the **UCS**, we make decisions based on the path cost of each action. The path cost is calculated by minimizing the number of poured layers. In calculating the path, we consider the number of poured layers in each action, and we perform the action with the least path cost first.

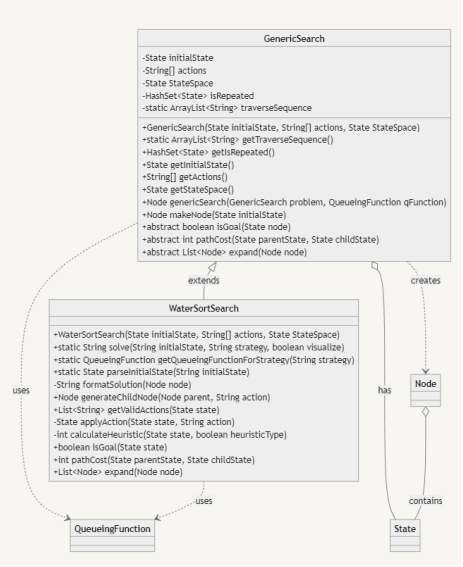
Detecting Repeated States

To detect repeating states, we implement a hash set and hash the state, including the layer and capacity of each bottle. If a repeated state is detected, we disregard it.

GenericSearch
is an abstract class that represents a problem. It has four attributes needed to implement a search problem: initial state, actions, a hash set to detect repeated states, and a traverse sequence to track the nodes expanded during the search.

GenericSearch has a **genericSearch** method, which is the basis of the generic search algorithm. It takes a generic problem, initializes it, and uses a queuing function to determine the search strategy. The specific queuing function is initialized according to the desired strategy.

It also has an abstract **isGoal** method for goalTest and an abstract **expand** method, which are used to get valid actions and perform them.



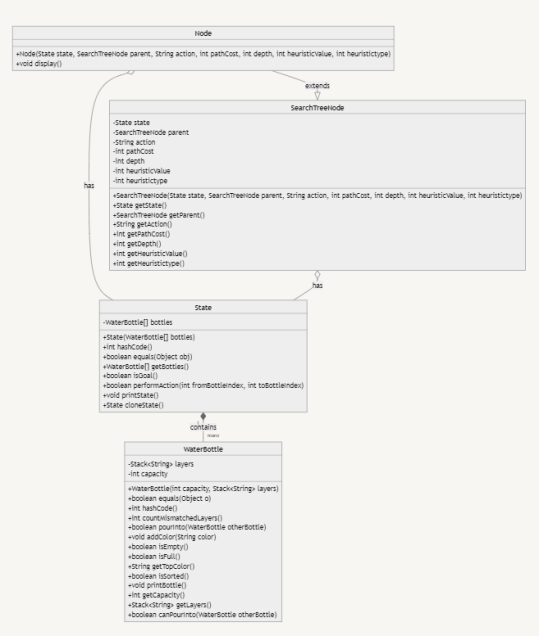
WaterSearch is the search problem of the project and inherits from the abstract class **GenericSearch**.

The main method is **solve**, which provides the solution to solve the problem. It takes the initial state and strategy. To solve the problem, the initial state is first formatted using a method called **formatSolution**, which parses the input and initializes the state and bottles for the first node in the search problem. Then, the **genericSearch** method, inherited from **GenericSearch**, is called with the initialized problem and the strategy. It initializes the queuing function and passes it to the generic search, which then provides the solution.

For the expansion, the **expand** method is overwritten from the abstract method in **GenericSearch**. It consists of two main methods:

- **validActions**, which generates the valid actions from the current state.
- **generateChildNode**, which takes the possible actions and generates new nodes with the specific state of bottles.

To generate the specific state, the **applyAction** method is used. It takes a state and an action, performs the action, and returns a new state.



State Class

- **Attributes:**
  - WaterBottle[] bottles: An array of **waterBottle** objects that represent the state of all the water bottles in the puzzle.
- **Methods:**
  - **State(WaterBottle[] bottles):** Constructor that initializes the state with a specific configuration of water bottles.
  - **hashCode()**: Generates a unique hash code for this state, useful for ensuring that the state can be uniquely identified when added to collections like hash sets.
  - **equals(Object obj)**: Checks if this state is equal to another, comparing their respective **waterBottle** configurations.
  - **getBottles()**: Returns the array of water bottles in the state.
  - i**sGoal()**: Checks whether the current state meets the goal conditions (likely all bottles being sorted).
  - **performAction(int fromBottleIndex, int toBottleIndex):** Performs an action to transfer water from one bottle to another.
  - **printState():** Prints the current state of the water bottles for debugging or display purposes.
  - **cloneState():** Returns a deep copy of the current state.

WaterBottle Class

- **Attributes:**
  - **Stack<String> layers:** Represents the layers of liquid inside the bottle, each string corresponding to a color.
  - int **capacity**: The maximum number of layers the bottle can hold.
- **Methods:**

- **WaterBottle(int capacity, Stack<String> layers)**: Constructor to create a water bottle with a specific capacity and initial layers of liquid.
- **equals(Object o):** Determines whether two water bottles are equivalent by comparing their contents.
- **hashCode():** Returns a unique hash code for the water bottle, useful for identifying the bottle in hash-based collections.
- **countMismatchedLayers()**: Counts how many layers in the bottle are not sorted (i.e., are not the same color).
- **pourInto(WaterBottle otherBottle)**: Attempts to pour water from this bottle into another bottle, ensuring that it follows the rules (e.g., the top colors must match, the other bottle must have space).
- **addColor(String color)**: Adds a new color to the top of the bottle.
- **isEmpty():** Checks if the bottle is empty.
- **isFull()**: Checks if the bottle is full.
- **getTopColor()**: Returns the color at the top of the bottle.
- **isSorted()**: Returns true if the bottle contains a single color or is empty.
- **printBottle()**: Prints the contents of the bottle.
- **getCapacity():** Returns the bottle's capacity.
- **getLayers():** Returns the stack of layers in the bottle.
- **canPourInto(WaterBottle otherBottle)**: Checks if it’s possible to pour liquid into another bottle.

SearchTreeNode Class

- **Attributes:**
  - **State state:** The current state of the puzzle at this node.
  - **SearchTreeNode parent:** The parent node, which represents the previous state.
  - **String action:** Describes the action taken to reach this node (e.g., pouring from one bottle to another).
  - **int pathCost:** The total cost to reach this node from the start (often related to the number of moves made).
  - **int depth:** The depth of the node in the search tree (how many moves deep it is).
  - **int heuristicValue**: The heuristic value, used to estimate the cost to reach the goal from this node.
  - **int heuristictype:** The type of heuristic used (could be multiple heuristic strategies for A*).
- **Methods:**
  - **SearchTreeNode(State state, SearchTreeNode parent, String action, int pathCost, int depth, int heuristicValue, int heuristictype):** Constructor for initializing a search tree node with the necessary information.
  - **getState():** Returns the state associated with the node.

- **getParent():** Returns the parent node.
- **getAction():** Returns the action that led to this node.
- **getPathCost():** Returns the path cost up to this node.
- **getDepth():** Returns the depth of this node in the search tree.
- **getHeuristicValue():** Returns the heuristic value.
- **getHeuristictype():** Returns the type of heuristic used.

Node Class

- This class extends **searchTreeNode** and might add more specialized functionalities for nodes in the search tree.

CPU Utilization and RAM Usage

```
BF
CPU time used (ms): 15 ms
Wall-clock time used (ms): 12 ms
CPU Utilization: 7.89 %
Memory used (MB): 3 MB
--------------------------------------------------------------------------------
DF
CPU time used (ms): 0 ms
Wall-clock time used (ms): 3 ms
CPU Utilization: 0.00 %
Memory used (MB): 4 MB
--------------------------------------------------------------------------------
IR
CPU time used (ms): 15 ms
Wall-clock time used (ms): 16 ms
CPU Utilization: 5.94 %
Memory used (MB): 9 MB
--------------------------------------------------------------------------------
UC
CPU time used (ms): 15 ms
Wall-clock time used (ms): 5 ms
CPU Utilization: 18.88 %
Memory used (MB): 10 MB
--------------------------------------------------------------------------------
GR1
CPU time used (ms): 0 ms
Wall-clock time used (ms): 1 ms
CPU Utilization: 0.00 %
Memory used (MB): 10 MB
--------------------------------------------------------------------------------
GR2
CPU time used (ms): 0 ms
Wall-clock time used (ms): 0 ms
CPU Utilization: 0.00 %
Memory used (MB): 10 MB
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
AS1
CPU time used (ms): 0 ms
Wall-clock time used (ms): 1 ms
CPU Utilization: 0.00 %
Memory used (MB): 10 MB
--------------------------------------------------------------------------------
AS2
CPU time used (ms): 0 ms
Wall-clock time used (ms): 0 ms
CPU Utilization: 0.00 %
Memory used (MB): 11 MB
--------------------------------------------------------------------------------
```

RAM Usage

- **BF** (3 MB): BF uses less memory in this case because the state space may not be very large, but in general, BF uses significant memory as it stores all nodes at the current depth.
- **DF** (4 MB): DF uses slightly more memory than BF, though it generally requires less memory compared to BF as it only needs to store nodes along the current search path.
- **IR** (9 MB): IR uses more memory because it essentially performs multiple depth-first searches, storing states for every depth iteration.
- **UC** (10 MB): UC typically uses more memory than BF and DF because it needs to store the frontier in memory, and it sorts nodes by their path costs.
- **GR1, GR2, AS1, AS2** (10-11 MB): These algorithms use similar memory because they store heuristically guided paths, and in A* they also maintain both cost and heuristic values,

increasing memory usage

CPU Utilization

- **GR1, GR2, AS1, AS2**: All these algorithms show **0% CPU utilization**, which likely indicates that they quickly reached a result without extensive node expansion, possibly because the heuristics guided the search efficiently.
- **UC**: CPU utilization of **18.88%**, relatively high because UC involves sorting the frontier based on path costs.
- **IR**: CPU utilization of **5.94%**, moderate due to the repeated depth-first search iterations.
- **DF**: CPU utilization of **0%**, possibly because the depth of the problem was small or the search terminated quickly without significant exploration.
- **BF**: CPU utilization of **7.89%**, moderate as expected for BF, which explores nodes level-by-level.

. 1. Completeness and Optimality

- **Breadth-First Search (BF)**:
  - **Completeness**: Complete
  - **Optimality**: Not optimal
- **Iterative Deepening (IR)**:
  - **Completeness**: Complete
  - **Optimality**: Not optimal
- **Greedy Search (GR1, GR2)**:
  - **Completeness**: Not complete
  - **Optimality**: Not optimal
- **Depth-First Search (DF)**:
  - **Completeness**: Incomplete
  - **Optimality**: Not optimal
- **Uniform Cost (UC)**:
  - **Completeness**: Complete.
  - **Optimality**: Optimal.
- *A* Search (AS1, AS2):
  - **Completeness**: Complete
  - **Optimality**: Optimal

Heuristics:

Heuristic 1: Count bottles that are not completely sorted
This heuristic counts the number of bottles that are not sorted. For each bottle, if it contains layers of mixed colors, it adds 1 to the heuristic value.

-Why this heuristic is admissible:
The number of unsorted bottles is a lower bound on the number of moves to reach the goal state. Since each unsorted bottle will require at least one action-an actual pour-to make a step towards being sorted. The heuristic is inconsiderate of the intricateness of the actions, yet it is not overly optimistic since all of the unsorted bottles will have to be taken care of.
-Proof of admissibility:

Because the heuristic counts only the number of bottles that are unsorted, and each of those bottles must be sorted in order to reach the goal, this heuristic is an underestimate or, in the best case, it is the correct number of bottles that need attention. Therefore, it never overestimates the number of actions required.

Heuristic 2: Count color transitions between layers
This is a heuristic that counts the number of color transitions between adjacent layers in a bottle.

For every consecutive layer in a bottle of different colors, it increments the heuristic value by 1.

-Why this heuristic is admissible:
Every transition, for bottles of a different color, represents a state that needs to eventually be resolved, in order to fulfill the goal of having bottles with uniform color layers. The heuristic estimates the minimum number of moves to remove transitions by pouring from/to other bottles.
-Proof of admissibility:

This heuristic counts transitions which need to be fixed to capture the difficulty of sorting. The minimum number of transitions provides a lower bound on the number of pours required to sort the bottle. Thus, this heuristic will not overestimate the true cost to solve the problem and therefore is admissible because this is a very conservative estimate that never results in a value higher than reality.

Implementation of Search algorithms:

1)BFS:

How BFS is implemented:

- **Queue Initialization**: The queue is implemented using a LinkedList, which allows efficient insertion at the end and removal from the front, crucial for BFS.
- **Enqueueing (enqueue)**: New nodes are added to the back of the queue, maintaining the BFS order by processing nodes level by level.
- **Dequeuing (dequeue)**: The node at the front of the queue (the oldest) is removed and returned for processing, ensuring BFS explores all nodes at the current level before moving to the next.
- **Queue Check (isEmpty)**: This method checks if there are more nodes to process, which signals when BFS is complete.
- **Display Function**: The displayQueue() method prints the contents of the queue, useful for debugging or understanding the queue's state during execution.

2)DFS:

How DFS is implemented:

- **Stack Initialization**: The stack is implemented using a LinkedList, which allows efficient insertion and removal from the top, crucial for DFS.
- **Enqueueing (enqueue)**: New nodes are pushed onto the top of the stack, ensuring that the most recently added nodes are explored first (LIFO behavior).
- **Dequeuing (dequeue)**: The node at the top of the stack (the most recent) is removed and returned for processing, which maintains the DFS approach of diving deeper into one branch before backtracking.
- **Stack Check (isEmpty)**: This method checks if the stack is empty, signaling when DFS has explored all nodes.
- **Display Function (displayQueue)**: This function is not implemented yet, but it would typically print the contents of the stack for debugging or to visualize the current state during execution

3)UCS:

How UCS is implemented:

- **Priority Queue Initialization**: The priority queue is initialized using a PriorityQueue, which automatically sorts nodes based on their path cost. The comparator is set to compare the nodes based on their path cost using SearchTreeNode::getPathCost.
- **Enqueueing (enqueue)**: Nodes are added to the priority queue, where they are automatically sorted by their path cost, ensuring that the node with the lowest cost is always at the front of the queue.
- **Dequeuing (dequeue)**: The node with the lowest path cost is removed and returned for processing, following UCS's goal of expanding the least-cost node first.
- **Queue Check (isEmpty)**: This method checks if the priority queue is empty, signaling when UCS has explored all nodes or paths.
- **Display Function**: The displayQueue() method prints the contents of the queue, useful for debugging or understanding the queue's state during execution.

4)IDS:

How IDS is implemented:

- **Stack Initialization**: The stack is implemented using a LinkedList to simulate a stack structure (LIFO), allowing efficient depth-first exploration at each iteration.
- **Depth Limit Handling**: The class has a currentDepthLimit variable, which controls the maximum depth allowed in each iteration of the search. Nodes are only enqueued (added to the stack) if their depth is within this limit.
- **Enqueueing (enqueue)**: Nodes are added to the stack only if their depth is less than or equal to the current depth limit. This ensures that DFS explores only up to the allowed depth during each iteration.
- **Dequeueing (dequeue)**: The node at the top of the stack (the most recent) is removed and returned for processing, maintaining the DFS approach of diving deeper into one branch before backtracking.
- **Depth-Limited Search**: The depthLimitSearch method performs a DFS that halts if the depth exceeds the current limit. It checks if the current node is a goal state or continues exploring its neighbors.
- **Iterative Deepening**: The iterativeDeepeningSearch method increments the depth limit after each DFS pass and clears the visited set, effectively restarting the search with a deeper exploration until the solution is found.
- **Queue Check (isEmpty)**: This method checks if the stack is empty, signaling when all nodes at the current depth limit have been explored.
- **Display Function**: The displayQueue() method prints the contents of the queue, useful for debugging or understanding the queue's state during execution.

5)Greedy:

How Greedy is implemented:

- **Priority Queue Initialization**: The priority queue is initialized using a PriorityQueue, which sorts nodes based on their heuristic values. The comparator is set to compare nodes by their heuristic value plus path cost.

- **Enqueueing (enqueue)**: Nodes are added to the priority queue, and they are automatically sorted based on their heuristic values. Greedy Best-First Search prioritizes nodes that are closest to the goal based on the heuristic estimate.
- **Dequeuing (dequeue)**: The node with the lowest heuristic value is removed and returned for processing. This ensures that the search focuses on exploring the most promising nodes first, based on the heuristic.
- **Queue Check (isEmpty)**: This method checks if the priority queue is empty, signaling when all nodes have been explored or when no more promising nodes are left.
- **Display Function**: The displayQueue() method prints the contents of the queue, useful for debugging or understanding the queue's state during execution.

6)A*:

How A* is implemented:

- **Priority Queue Initialization**: The priority queue is initialized using a PriorityQueue, which sorts nodes based on their **cost function**. The cost function is a combination of the node's path cost (g(n)) and its heuristic value (h(n)), where nodes are ordered by g(n) +h(n). This helps prioritize nodes that have a lower estimated total cost.
- **Enqueueing (enqueue)**: Nodes are added to the priority queue, and they are automatically sorted based on the sum of their path cost and heuristic value (g(n) +h(n)). This ensures that A* explores nodes that are both cheaper to reach and seem promising based on the heuristic.
- **Dequeuing (dequeue)**: The node with the lowest g(n) +h(n) value (i.e., the most promising node with the lowest estimated total cost) is removed and returned for processing. This keeps A* focused on finding the optimal path to the goal.
- **Queue Check (isEmpty)**: This method checks if the priority queue is empty, which would signal that there are no more nodes to explore or that the goal has been reached.
- **Display Function**: The displayQueue() method prints the contents of the queue, useful for debugging or understanding the queue's state during execution.

References

we used Notion for writing this report and making the class diagram

https://www.notion.so/

Note: please zoom in for the diagram as we make it small so the report will not exceed 10 pages ;)

Thank you for reading! 😊📚🎓