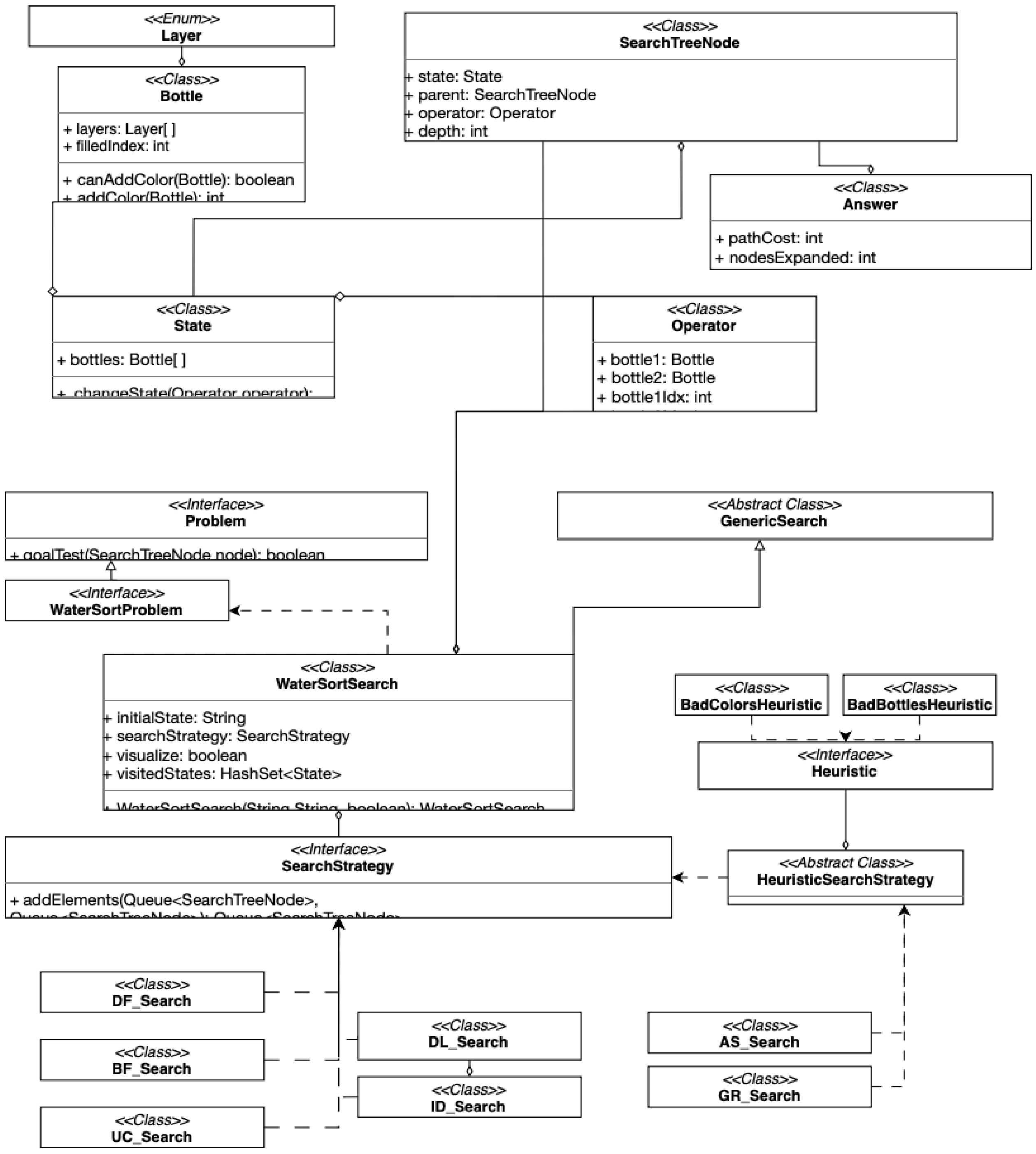
Water Sort Puzzle Search Agents

Class Diagram:



Search Algorithms Implementation Details

The difference between each search strategy and the other is how they put the search tree node(s) in their data structure and which node they pick to be further expanded. So each search algorithm i.e DFS, BFS, IDS, etc implements the search strategy interface which requires each search algorithm to implement 4 main things:

SearchStrategy interface

```
Queue<SearchTreeNode> generateDataStructure();
Queue<SearchTreeNode> addElements(Queue<SearchTreeNode> oldNodes,
Queue<SearchTreeNode> newNodes);
SearchTreeNode getElement(Queue<SearchTreeNode> elements);
SearchTreeNode runSearchAlgorithm(Problem problem);
```

1) what is the data structure to store the search tree nodes
2) how to add elements to this data structure
3) how to get elements from this data structure
4) how to run the search algorithm

Depth First Search:

1. Returns a double ended queue
2. Add to the front of the queue
3. Get from the front of the queue
4. Call the `General_Search` **function** in the `GenericSearch` **class** with the problem and myself(DFS **search strategy** in this case)

Breadth First Search:

1. Returns a double ended queue
2. Add to the end of the queue
3. Get from the front of the queue
4. Call the `General_Search` **function** in the `GenericSearch` **class** with the problem and myself as the **search strategy**

Iterative Deepening Search:

1. Returns a double ended queue
2. Add to the end of the queue
3. Get from the front of the queue

4. Call the `General_Search` **function** in the `GenericSearch` **class** with the problem and `DL_Search`(Depth Limited Search) as the **search strategy** inside a loop and increase the depth until you find a solution

runSearchAlgorithm function implementation inside the ***ID_Search* class**

```
@Override
public SearchTreeNode runSearchAlgorithm(Problem problem) {
   int depthLimit = 0;
   SearchTreeNode node;

   while (true){
       DL_Search depthLimitedSearch = new DL_Search(depthLimit);
       node = depthLimitedSearch.runSearchAlgorithm(problem);

       if(node!=null)
           break;

       depthLimit++;
       ((WaterSortSearch) problem).visitedStates = new HashSet<>();
   }
   return node;
}
```

Uniform Cost Search:

1. Returns a priority queue and priority is based on the search tree node with lowest cost from root
2. Add to the priority queue
3. Remove node with the least cost from the priority queue
4. Call the `General_Search` **function** in the `GenericSearch` **class** with the problem and myself as the **search strategy**

Greedy Search:

All steps are same as UCS except step 1 in which we return a priority queue where the priority is based on the search tree node with the lowest heuristic value

A* Search:

All steps are same as UCS except step 1 in which we return a priority queue where the priority is based on the search tree node with the lowest (heuristic value + cost from root)

Heuristics

We employed two heuristic functions. Either one of them could be used to calculate the heuristic value(s) when running either Greedy or A* search.

First Heuristic (Bad Bottles Heuristic)

Description

The heuristic value is calculated by counting the number of bottles that contain number of different colors greater than 1

Heuristic Function Implementation:

```
@Override
public int estimateCostToGoal(State state) {
   int badBottles = 0;
   Bottle [] bottles = state.getBottles();
   for(Bottle bottle: bottles){
       badBottles+= bottle.isBottleSameColor()? 1 : 0;
   }

   return badBottles;
}
```

Admissibility

This heuristic will never be an overestimate for the cost from a certain state to the goal state because for a state to reach the goal it will at least need to do the pour operation a number of times equal to the number of bad bottles. It is easy to see that you can not fix *N* bad bottles in less than *N* operations.This is an underestimate from the real cost since we are assuming that one pour operation fixes a bad bottle which in reality could be much more than one operation. You can not fix two bad bottles using one pour operation.

Heuristic2 (Number of bad colors heuristic)

Description

The heuristic value is calculated by counting how many different colors are in each bottle and summing these values over all bottles. A bottle with two colors will have a value of 2 and a bottle that contains one or zero colors will have a value of 0.

Heuristic Function Implementation:

```java
@Override
public int estimateCostToGoal(State state) {
   int badColors = 0;
   Bottle [] bottles = state.getBottles();
   for(Bottle bottle: bottles){
       badColors += bottle.numberOfDifferentColors();
   }
   return badColors;
}
```

```java
public int numberOfDifferentColors(){
   HashSet<Layer> numberOfDiffLayers= new HashSet<>();

   for(int layer =filledIndex; layer>-1; layer--){
       numberOfDiffLayers.add(layers[layer]);
   }
   return Math.max(0, numberOfDiffLayers.size()-1);
}
```

Admissibility

The admissibility of this heuristic is similar to the bad bottles heuristic since if for each bottle we calculate the number of different colors we can not reach a cost to a goal state that is less than this heuristic value. If we have a bottle with M different colors, it is impossible to make this bottle contain only one color in less than M-1 pour operations. If this is true for one bottle then it is true for all bottles. So we sum the number of different colors across N bottles and this is the heuristic value. This is underestimate for the real cost because each pour operation can have a cost of at least 1 and here we ignore the number of layers of each color and just assume it to be 1.
This heuristic is better than the previous heuristic since it is much closer to the real cost than the previous heuristic.

Search Algorithms Performance Comparison

Test Case:

```
StringBuilder testCase9Input = new StringBuilder("5;4;" +"b,y,r,b;" +
"b,y,r,r;" +"y,r,b,y;" + "e,e,e,e;" + "e,e,e,e");
```

DFS

Answer:
pour_2_4,pour_2_3,pour_1_2,pour_4_1,pour_3_4,pour_2_3,pour_1_2,pour_4_1,pour_3_4,pour_2_3,pour_4_2,pour_3_4,pour_1_3,pour_4_1,pour_3_4,pour_2_3,pour_4_2,pour_3_4,pour_0_4,pour_2_3,pour_1_2,pour_4_1,pour_3_4,pour_2_3,pour_4_2,pour_3_4,pour_4_0,pour_2_3,pour_1_2,pour_4_1,pour_3_4,pour_2_3,pour_4_2,pour_0_4,pour_2_0,pour_4_1,pour_3_4,pour_2_3,pour_4_2,pour_3_4,pour_1_3,pour_4_1,pour_3_4,pour_2_3,pour_4_2,pour_3_4,pour_0_3;112;168

CPU Utilization	RAM Usage	Expanded Nodes
11.23 %	5 MB	168

BFS

Answer:pour_0_3,pour_0_4,pour_1_3,pour_1_4,pour_0_1,pour_2_4,pour_2_1,pour_2_0;8;877

CPU Utilization	RAM Usage	Expanded Nodes
9.76 %	7 MB	877

IDS

Answer:
pour_2_4,pour_2_3,pour_1_2,pour_0_2,pour_0_4,pour_0_3,pour_2_0,pour_1_2;10;221

CPU Utilization	RAM Usage	Expanded Nodes
10.94 %	8 MB	221

UCS

Answer: pour_0_4,pour_0_3,pour_1_4,pour_1_3,pour_0_1,pour_2_3,pour_2_1,pour_2_0;8;684

CPU Utilization	RAM Usage	Expanded Nodes
11.50 %	7 MB	684

GR1

Answer:
pour_0_3,pour_2_4,pour_1_3,pour_1_0,pour_0_4,pour_0_2,pour_3_0,pour_2_1,pour_2_0;12;6
4

CPU Utilization	RAM Usage	Expanded Nodes
9.35 %	5 MB	64

GR2

Answer: pour_1_3,pour_1_4,pour_0_3,pour_0_4,pour_0_1,pour_2_4,pour_2_1,pour_2_0;8;36

CPU Utilization	RAM Usage	Expanded Nodes
8.82 %	5 MB	36

A*1

Answer: pour_1_4,pour_1_3,pour_2_3,pour_2_1,pour_2_4,pour_0_4,pour_0_2,pour_0_1;8;348

CPU Utilization	RAM Usage	Expanded Nodes
9.47 %	6 MB	348

A*2

pour_1_4,pour_1_3,pour_2_3,pour_0_4,pour_0_3,pour_2_1,pour_0_1,pour_2_0;8;188

CPU Utilization	RAM Usage	Expanded Nodes
10.46 %	6 MB	188

Analysis of Results

The DFS search strategy had the second highest CPU utilization and the lowest RAM usage. It has a high CPU utilization because it explores depth first and if there is an answer on a different branch it could take a lot of time to reach this answer. The lowest RAM usage is also supported by the fact that it had the lowest number of nodes expanded.It is not optimal as evidenced by the cost of the answer which is 112 (the lowest possible cost is 8). It is complete since we keep track of the visited nodes to avoid cycles or being stuck in a branch so eventually we will reach an answer if there is one.

BFS had a better CPU utilization than DFS but a higher RAM usage. It had a better CPU utilization because it goes through the search tree level by level avoiding being stuck in a bad branch for a long time. It had a higher number of RAM usage most probably because it had a higher number of expanded nodes. It is also complete because we do not visit any states we expanded before however it is not optimal. It returned an answer with the lowest cost but this was by luck.

IDS had a better CPU utilization than DFS but worse than BFS. This is because it does many depth first searches with increasing depth limit until a solution is found. It had more number of expanded nodes than DFS but less than BFS because it goes deep first and found the answer earlier while BFS has to finish the whole level before going to the next one.However, it has a higher RAM usage than both DFS and BFS; this is most probably because memory is not freed between every run of a depth limited search. IDS is complete but not optimal as evidenced by its answer.

UCS had the highest CPU utilization because it aims to find the best solution sacrificing efficiency. It has a RAM usage same as BFS and a number of expanded nodes just below BFS. This is because it tries to explore the best nodes and it does not go into bad branches aimlessly so it has a comparable performance in RAM usage and number of nodes expanded to BFS. UCS is both complete and optimal evidenced by its result.

In GR$_{\text{(first heuristic)}}$, it has the second lowest amount of expanded nodes and the lowest amount of RAM usage because it aims to find the closest solution but not an optimal one. This is also evidenced by the CPU utilization, in which GR has the second lowest CPU utilization among all other search strategies.

In GR$_{\text{(second heuristic)}}$, it has the lowest amount of expanded nodes and the lowest amount of RAM usage because it aims to find the closest solution but not an optimal one. This is also evidenced by the CPU utilization, in which GR has the lowest CPU utilization among all other search strategies. The second heuristic is much more robust than the first one since it is a closer estimate to the actual cost.
GR is not optimal evidenced by its results but it is complete

In A*$_{\text{(first heuristic)}}$, it had a higher CPU utilization and RAM usage than GR1 and GR2 as well as more number of expanded nodes. This is because it tries to be both optimal and efficient. A*$_{\text{(first heuristic)}}$ is both optimal and complete because the heuristic function is admissible and cost function is >0.

In A*$_{\text{(second heuristic)}}$ had a higher CPU utilization than A*$_{\text{(first heuristic)}}$ even though it had a fewer number of expanded nodes.. This is because the second heuristic requires more calculation.
It had a fewer number of expanded nodes than A*$_{\text{(first heuristic)}}$ because the second heuristic is much more of a closer estimate to the actual cost to goal than the first heuristic function. It also had same RAM usage as A*$_{\text{(first heuristic)}}$ because the difference between the number of expanded nodes between them is not significant. A*$_{\text{(second heuristic)}}$ is both optimal and complete because the heuristic function is admissible and cost function is >0.