CSEN 901: Introduction to Artificial Intelligence
Projct 1: Water Sort

Project Report

Abdelrahman Aboelkehir, Hossain Ghoraba,
Mahmoud Aboelenein, Omar Hesham

18.10.2024

1 Introduction

This project is the implementation of a search agent to solve the water sort puzzle. You can find a description of the water sort puzzle in the document titled **Project Description.pdf**. The agent is implemented in Java. The search the agent performs does not prune any branches early, nor does it perform any checks for whether the current node can or can not lead to a solution. A branch in the search is terminated only when there are no more possible nodes to expand from the leaf of that branch (in this probelm, this means there are no possible pour combinations to apply to the current state). The search terminates when either **a.** A node selected for expansion is identified as a goal state, or **b.** No more nodes on the search frontier can be expanded.

2 Program Running Cycle

For the Water Sort Problem, running begins in the `WaterSortSearch` class. The class has a method `solve` that takes as an input a String `initialState`, which contains the initial configuration of the bottles and the layers within them, as well as the number of bottles and their capacities, encoded in a specified string format. It also takes a `strategy` specifying the search algorithm to use, and a boolean `visualize` that enables or disables printing various information about the run to the console. From there, the initial state is pares into our `WaterSortState` structure using a parsing method in `WaterSortUtils`, a new `WaterSortProblem` is created using the initial state and the search strategy, and `executeSearchStrategy(WaterSortProblem waterSortProblem, String strategy, boolean visualize`.

`executeSearchStrategy()` is a method in the `SearchStrategy` class that:

1. Gets the appropriate queuing function for the search algorithm based on the strategy
2. runs the corresponding `Search()` method, which itself just runs `generalSearch()` in the `GenericSearch` class.

For example, here is the method for Breadth-First search:

```
public static Node BreadthFirstSearch(Problem problem,boolean visualize) throws
CloneNotSupportedException {
  return GenericSearch.generalSearch(problem, new EnqueueAtEnd(), visualize);
}
```

The return of this method is the node corresponding to a solution state (or `null`, if no solution is found).

3 Project Hierarchy

Here is a hierarchy of all the project files:

```
src
└── main
    └── java
        ├── code
        ├── generic
        │   ├── QueueingFunctions
        │   │   ├── QueueingFunction <<Interface>>
        │   │   ├── AStar1QueueingFunction
        │   │   ├── AStar2QueueingFunction
        │   │   ├── EnqueueAtEnd
        │   │   ├── EnqueueAtFront
        │   │   ├── EnqueueAtFrontWithDepthLimit
        │   │   ├── GREEDY1QueueingFunction
        │   │   └── GREEDY2QueueingFunction
        │   ├── FixedSizeStack
        │   ├── Operator <<Interface>>
        │   ├── OperatorResult
        │   ├── Problem
        │   └── SearchState
        │       └── OrderedInsert
        ├── utils
        │   ├── ConditionalPrintStream
        │   ├── Methods
        │   └── WaterSortUtils
        ├── watersort
        │   ├── Bottle
        │   ├── Color
        │   ├── LayerGroup
        │   ├── Pour
        │   ├── WaterSortProblem
        │   ├── WaterSortState
        │   └── WaterSortUtils
        ├── GenericSearch
        ├── Node
        └── WaterSortSearch
```

4 Class Diagram

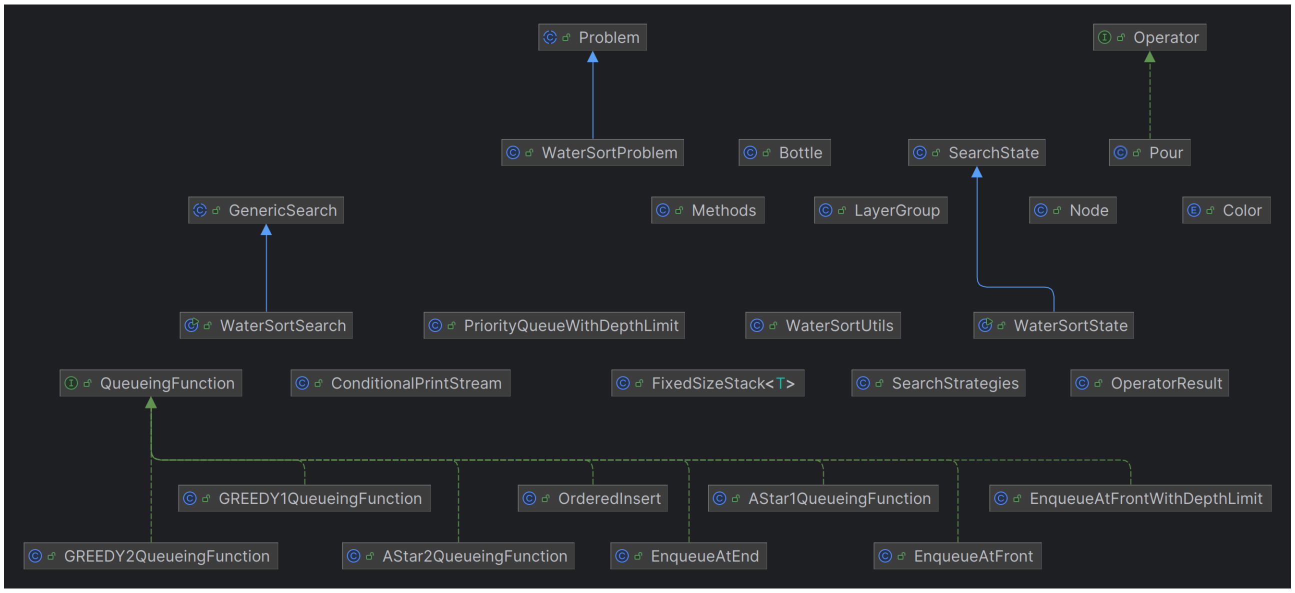

Figure 1: Class Diagram

5 Implementation of the Search Algorithms

Our implementation for all search algorithms closely follows the description of the *GeneralSearch* algorithm outlined in lecture 2 of the course:

Algorithm 1 GENERAL-SEARCH(problem, QING-FUN)

```
1: return a solution, or failure
2: $nodes \leftarrow$ MAKE_Q(MAKE_NODE(INIT_STATE($problem$)))
3: while true do
4:     if $nodes$ is empty then
5:         return failure
6:     end if
7:     $node \leftarrow$ REMOVE_FRONT($nodes$)
8:     if GOAL_TEST($problem$)(STATE($node$)) then
9:         return node
10:    end if
11:    $nodes \leftarrow$ QING_FUN($nodes$, EXPAND($node$, OPER($problem$)))
12: end while
```

All the algorithms use a Java `PriorityQueue` as their queue. The `PriorityQueue` in Java orders objects based on a given `Comparator`. The classes in the `QueueingFunctions` package create and return a `PriorityQueue` with the appropriate `Comparator`, which will then be used to hold and order all the nodes during the search.

For example, here is the queuing function for Breadth-First Search:

```java
public class EnqueueAtEnd implements QueuingFunction {
  @Override
  public PriorityQueue<Node> apply() {
    return new PriorityQueue<>(Comparator.comparingInt(Node::getDepth));
  }
}
```

In this case, a `PriorityQueue` is created using a comparator that says to order the nodes by depth in ascending order. Consequently, all the nodes in this queue will be ordered in ascending order of depth.

Here is a comprehensive list of the way by which the `PriorityQueue` orders nodes within it for every search algorithm we have implemented:

1. Breadth-First Search (BFS): Depth (ascending, i.e from *shallowest* to *deepest*)
2. Depth-First Search (DFS): Depth (descending, i.e from *deepest* to *shallowest*)
3. Iterative Deepening Depth-First Search (IDS): Same as DFS
4. Uniform-Cost Search (UCS): Path cost from the root to the node `(f(n) = g(n))`(ascending)
5. Greedy Best-First Search: The value of the heuristic function `(f(n) = h(n))`(we have two **admissible** heuristics we can use)
6. A* Search: The value of the path from the root plus the heuristic value for the current node `(f(n) = g(n) + h(n))`

Here is the main body of how searching is done (in the `GenericSearch` class):

```
public static Node generalSearch(Problem problem, QueuingFunction queuingFunction, boolean visualize)
            throws CloneNotSupportedException {
        nodesExpanded = 0;
        int nodesVisited = 0;
        Node solutionNode = null;
        expandedStates.clear();
        candidateSolutions.clear();

        Node initialNode = makeNode(problem.getInitialState());
        PriorityQueue<Node> nodes = queuingFunction.apply();
        nodes.add(initialNode);

        while (!nodes.isEmpty()) {
            Node currentNode = removeFront(nodes);
            currentNode.setOrderOfVisiting(nodesVisited++);
            if (problem.goalTestFn(currentNode)) {
                solutionNode = currentNode;
                break;
            }
            List<Node> expandedNodes = expand(currentNode, problem.getOperators(), problem);
            nodes.addAll(expandedNodes);
        }

        return solutionNode;
    }
```

5.0.1 Goal Test

One important thing to mention is that the goal test is applied to a node first when it is selected for expansion, not when it is created. This has the unfortunate side-effect of increasing the time complexity of DFS from $O(b^d)$ to $O(b^{d+1})$, because an extra level (at least in the worst case) will be created beyond the goal node before the goal node is selected for expansion and discovered to be a goal, as well as perhaps increasing the time complexity of the rest of the search algorithms (by a factor I am not entirely sure about), but has the fortunate side-effect of making UCS optimal (if the goal test was applied to a node when it is first created, UCS would not be optimal). If you are curious, you can find a small discussion of why this may be the case in Russel & Norvig, section 3.4.2

5.1 Heuristic Functions

We have mentioned earlier that we have two heuristic functions available for use. In this section, we will examine those two heuristic functions and argue why they may be admissible.

Heuristic 1: The number of bottles where the layers are not all the same color.

It is easy to show that this heuristic is admissible. For any state not to be a goal state, there must be at least 1 bottle where all the layers are not the same color. To make this state a goal state, you must pour at least 1 layer from one bottle to another. Therefore, this heuristic can never overestimate the cost (which is the number of layers poured) of the solution.

Heuristic 2: The difference between the current filled capacity of the bottle and the highest number of layers with the same color, all added up together.

For example, consider the following state with 3 bottles: `[e,e,r,g,r]`, `[e,b,r,b,b,y]`, `[e,e,e,e,e]`. The value of the heuristic function for this state will be: $(3-2)+(5-3)+(0)=3$

It is also not too difficult to show that this heuristic is admissible. If the node is a goal state (either all the bottles are empty, or all the bottles have layers of the same color), its value is 0. For a state that is only one layer away from a goal state, such as `[y,y,y,y,y]`, `[e,g,b,b,b]`, `[e,g,g,g,g]`, the value of the heuristic is $(0)+(1)+(0)=1$. And this is in the "best-case" scenario where the layer to be poured is directly on top. If the layer(s) was/were anywhere other than on top, it would certainly take more than 1 pour operation to reach a goal state. Notice that we only calculate the difference between the filled capacity of the bottle and the number of the most repeated color layer.

The value of this heuristic for an empty bottle is therefore 0 (current capacity: 0 - number of most repeated color: 0, = 0.)

5.2 Implementation of Iterative-Deepening Search

Iterative-Deepening Search is special in that it is the only search algorithm that works differently from the remaining search algorithms. For the remaining search algorithms, nodes are simply added to the `PriorityQueue`, removed from the front of the queue, and expanded, until either the queue is empty (there are no more nodes left to expand), or a solution node has been found. IDS works differently in that it is the only search algorithm we have implemented that searches the whole tree from the root node more than once.

```
public static Node IterativeDeepeningSearch(Problem problem, boolean visualize){
  int infinity  = Integer.MAX_VALUE;

  for(int depth = 0; depth < infinity; depth++){
    Node solution = DepthLimitedSearch(problem, depth, visualize);
    if(solution != null){
      return solution;
    }
  }

  return null;
}
```

Our method for IDS runs DFS up to "infinity" (in Java, `Integer.MAX_VALUE` is equal to 2,147,483,647 which we will assume is good enough for our purposes.) up to the specified `depth`, staring from 0 and incrementing the depth by 1 every iteration. Every time `DepthLimitedSearch` is called, it starts a new search problem, unrelated to the search in the previous iteration.

`LimitedDepthSearch` is just DFS that runs up to a specified depth limit.

```
public static Node LimitedDepthSearch(Problem problem, int depth, boolean visualize)
throws CloneNotSupportedException {
    return GenericSearch.generalSearch(problem, new EnqueueAtFrontWithDepthLimit(depth), visualize);
  }
```

Its queuing function `EnqueueAtFrontWithDepthLimit` uses a custom data type that is a priority queue that only allows the addition of elements within a specified depth limit.

```
public class EnqueueAtFrontWithDepthLimit implements QueuingFunction {
  public int depthLimit;

  public EnqueueAtFrontWithDepthLimit(int depthLimit){
    this.depthLimit = depthLimit;
  }

  @Override
  public PriorityQueue<Node> apply() {
    return new PriorityQueueWithDepthLimit(Comparator.comparingInt(Node::getDepth).reversed(), depthL
  }
}
```

And here is how `PriorityQueueWithDepthLimit` is defined:

```
public class PriorityQueueWithDepthLimit extends PriorityQueue<Node> {
  private final int depthLimit;

  public PriorityQueueWithDepthLimit(int depthLimit) {
    super(); // Call to the default PriorityQueue constructor
    this.depthLimit = depthLimit;

  @Override
  public boolean addAll(Collection<? extends Node> nodes) {
    boolean modified = false;
    for (Node node : nodes) {
      if (node.getDepth() <= depthLimit) {
        modified |= super.add(node); // Add nodes only if within the depth limit
      }
    }
    return modified;
  }
}
  NOTE: Some code has been trimmed from this class for simplicity.
  You can view the full code in the corresponding class file in the project.
```