AI Project 1 Report

Search Algorithms:

General Search Algorithm:

In our implementation, the GeneralSearch class is where the search algorithms are implemented. The different searching strategies extend this class and define the abstract function `EVAL_Fn()` that changes the way we insert into the priority queue. They can reuse the general functions for node exploration, goal testing, and node expansion provided by the parent class. It includes the following strategies:

1. BFS: overrides the `EVAL_Fn()` method to use the queue as a first-in, first-out (FIFO) structure by prioritizing nodes based on their depth. It does so by returning `node.getDepth()`. BFS explores all nodes at depth `d` before moving to nodes at depth `d+1`.
2. DFS: overrides the `EVAL_Fn()` method to prioritize deeper nodes in the search tree. Unlike BFS, DFS behaves more like a last-in, first-out (LIFO) structure by exploring nodes that are deeper first. We do this by returning `-node.getDepth()`, which causes the priority queue to prioritize nodes with higher depth (d+1 prioritized over d), allowing DFS to explore as deep as possible.
3. Greedy: overrides the `EVAL_Fn()` method and implements two different heuristic functions: `H_n1()` and `H_n2()`. It makes decisions based on the lowest heuristic value, by trying to find the cheapest cost of reaching the solution.
    - Heuristic 1:
      This heuristic estimates the number of moves needed to solve the problem by calculating how many layers on each bottle need to be moved to match the correct color sequence. This is done by:
        - Look for the longest sequence of identical colors from the bottom of the bottle.
        - Computes how many layers remain after this sequence that need to be moved.

      This heuristic is more informed than the second heuristic because it considers the number of layers to be moved.
      Since the heuristic counts only the layers that must be moved and does not make any assumptions about unnecessary moves, it will **never overestimate** the number of required moves. Therefore, this heuristic is **admissible.**
      Example: bottle : H(top->[b, b, r, r, g, b*, b*, b*]<-bottom) = 8 - 3 = 5
    - Heuristic 2:
      This heuristic calculates the number of times the color changes within a bottle, which shows how many transitions are needed to solve the problem.
      This heuristic assumes that you can move multiple layers at once.
      It is **admissible** since it does not overestimate the number of actions required to solve the problem.
      Example: bottle : H(top->[b, b, r, r, g, b*, b*, b*]<-bottom) = 3, because: (b->r, r->g, g->b)

4. A*: extends the `GreedyS` class and overrides the `EVAL_Fn()` method to consist of both the **cost to reach the current node `g(n)`** and the **heuristic estimate to the goal `h(n)`.**
   In the A* implementation, the evaluation function `EVAL_Fn()` computes the value of `g(n) + h(n)`:
   `g(n)`: The actual path cost, retrieved by `node.getPathCost()`.
   `h(n)`: The heuristic value is calculated by calling `super.EVAL_Fn(node, version)`, which is the heuristic calculation implemented in `GreedyS` class.
5. IDS: In this implementation, the class `IDS` extends the `DFS` class and overrides the `GENERAL_SEARCH()` method to introduce the iterative deepening strategy.
   It is implemented by combining the depth-limited search approach of DFS with an iterative loop that gradually increases the maximum search depth. It starts with a depth limit of 1 and continues increasing this limit until a solution is found or the search reaches the maximum available depth. This allows it to achieve both the completeness and optimality of BFS while maintaining the space efficiency of DFS.
6. UCS: In the `UCS` class, the search prioritizes nodes with the lowest cumulative cost from the start node. UCS is a search algorithm that expands the node with the lowest path cost first. The method `EVAL_Fn()` returns `node.getPathCost()`, ensuring that the node with the lowest cost is expanded first. Path Cost is the total cost from the start node to the current node. It is optimal and complete for problems with non-negative path costs, guaranteeing that the least-cost path to the goal will be found.

Performance:

Algorithm	Optimality	Completeness
BFS	Optimal, if the path cost of the nodes is increasing by depth and from left to right (or order of nodes enqueue), so generally not optimal	Complete, will find a solution if it exists.
DFS	Not Optimal	Not Complete, can be stuck exploring an infinite branch
Greedy	Not Optimal	Not Complete
A*	Optimal, since the heuristic is admissible.	Complete, since g(n) is positive.
IDS	Not Optimal	Complete
UCS	Optimal, as it expands nodes in increasing order of path cost.	Complete

- Grid 0:

Algorithm	Memory	CPU Utilization	Expanded nodes
BFS	7MB	5.8%	13
DFS	7MB	5.9%	6
Greedy	7MB	5.53%	7
Greedy 2	7MB	7.26%	6
A*	7MB	4%	11
A* 2	7MB	5.05%	13
IDS	7MB	5.3%	44
UCS	7MB	4.8%	13

- Grid 1:

Algorithm	Memory	CPU Utilization	Expanded nodes
BFS	13MB	6.87%	536
DFS	7MB	6.26%	22
Greedy	7MB	5.33%	8
Greedy 2	7MB	5.67%	8
A*	7MB	7.18%	29
A* 2	8MB	3.97%	56
IDS	15MB	8.57%	780
UCS	10MB	6.08%	355

- Grid 2:

Algorithm	Memory	CPU Utilization	Expanded nodes
BFS	13MB	6.85%	536
DFS	7MB	9.54%	22
Greedy	7MB	6.89%	8
Greedy 2	7MB	7.66%	8

A*	7MB	6.26%	29
A* 2	8MB	4.16%	56
IDS	15MB	9.86%	780
UCS	10MB	10.93%	355

- Grid 3:

Algorithm	Memory	CPU Utilization	Expanded nodes
BFS	8MB	6.84%	72
DFS	7MB	7.68%	13
Greedy	7MB	5.32%	11
Greedy 2	7MB	5.01%	11
A*	8MB	5.67%	36
A* 2	8MB	5%	50
IDS	15MB	4.38%	462
UCS	8MB	10.70%	73

- Grid 4:

Algorithm	Memory	CPU Utilization	Expanded nodes
BFS	16MB	6.68%	532
DFS	8MB	4.5%	20
Greedy	7MB	7.35%	6
Greedy 2	10MB	4.55%	6
A*	8MB	7.14%	33
A* 2	10MB	4.3%	130
IDS	18MB	6.13%	668
UCS	16MB	3.71%	554

Comments:

Breadth First Search (BFS): We can see that the BFS, although it is complete, is the most memory intensive one (excluding IDS). On average it was one of the highest memory usages. In larger grids, BFS's memory usage spikes significantly (up to 16MB), reflecting the complexity of maintaining all nodes at each level (space complexity is $\mathbf{b^d}$**).** Expanded Nodes

vary depending on the depth of the solution and size of the grid ranging 13~536 nodes (which correlates with memory usage). CPU utilization varies from 5.8% to 6.87%, which is average.

Depth First Search (DFS): In the DF search, and after implementing states history, had the least memory footprint. Memory ranged between 7~8MB VS 16~18MB in the largest memory in other search strategies, which may record around 50% less memory. Regarding the number of expanded nodes, the numbers ranged from 6 to 22 but the solution usually had high cost and/or depth. Sometimes it's faster than others in finding "a" solution but not an optimal one. The CPU utilization for DFS varies from 4.5% to 9.54%, on average similar to the BFS due to the fact that we nearly prevented infinite and repeated states.

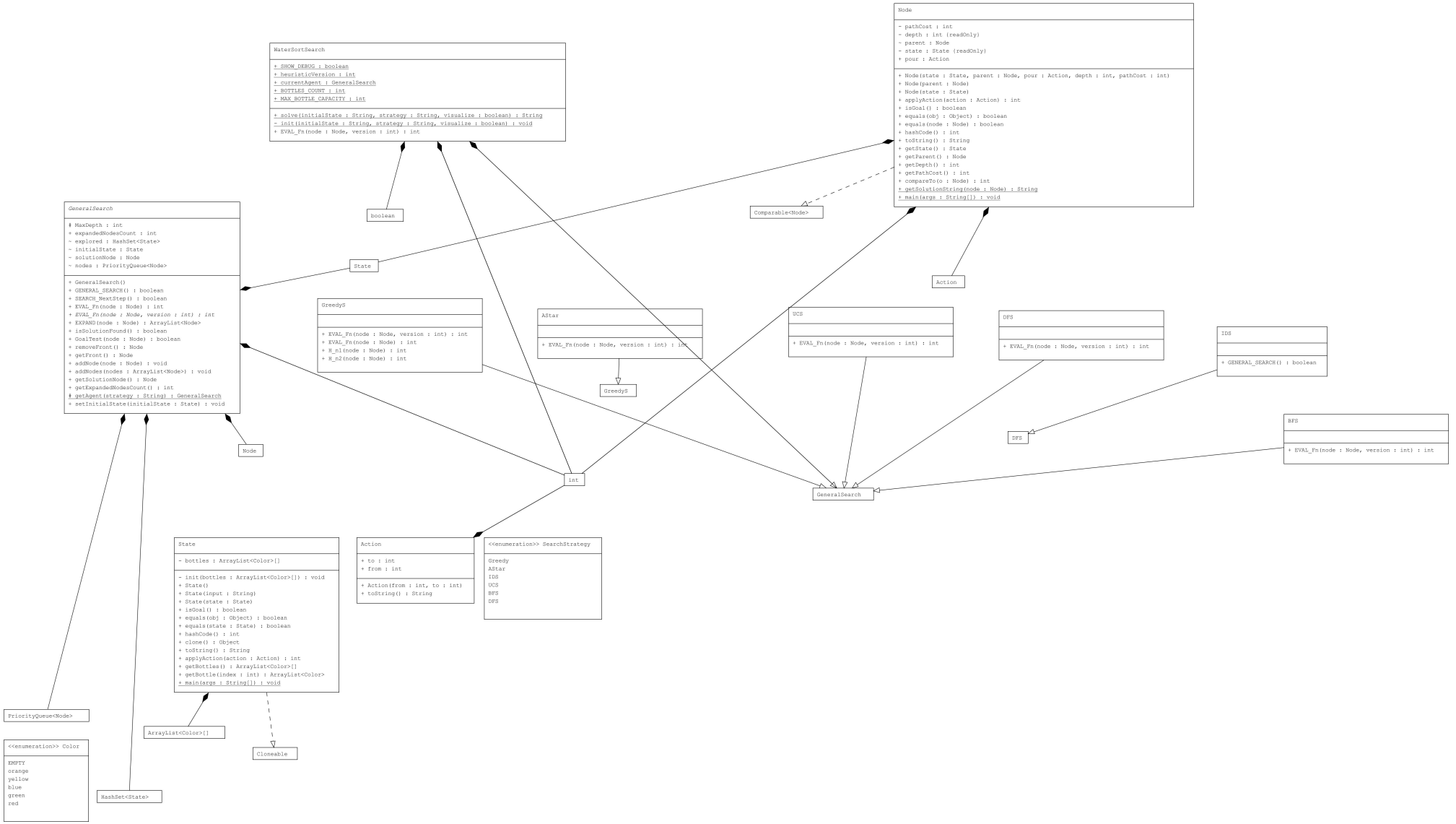
Uniform Cost Search (UCS): UCS focuses on the cumulative path cost. Memory Usage ranged from 7MB to 16MB. This trend of increase can also be noticed in the number of expanded nodes which was 13~554 nodes. Both depended on many factors like the actual cost of the smallest optimal solution, the depth of this optimal solution, and maybe the similarity of the path costs with each other. As we all know, BFS is a UCS with edges (or actions) cost of 1 or ***C***, this explains why it can sometimes find the solution quickly and sometimes needs a lot of expansions and memory. For the CPU Utilization, it ranges between 3.71% and 10.93%. The reason for this is, again, the nature of the search problem, and unlike the above BFS and DFS, the UCS needs more processing to keep the nodes sorted by their cost every time new expansions and insertions are made.

Iterative Deepening Search (IDS): ID is the search combining BF and DF. This ensures completeness and optimality but at a higher computational cost. Memory usage is noticed to increase (up to 18MB), reflecting the need to store nodes across iterations. The number of nodes reached up to 780 (the highest), for the same reason of the iterations. The memory usage depends on the depth of the nearest solution. CPU Utilization is above average, 4.38~9.86%, due to the need of trying, repeating and processing lower level nodes multiple times until the solution level.

Greedy Search: Greedy is known for its efficiency, especially after we used 2 admissible functions, due to the fact that it chooses nodes greedily. It consistently used around 7MB memory throughout the tests of both heuristic 1 and 2. Number of expanded nodes was the lowest among all other strategies (6 to 8). For sure the greedy search is not always optimal, but generally when the heuristic function is near the actual cost remaining the greedy will perform better and better expanding a path to goal faster. CPU utilization ranges from 4.55% to 7.66% (for both h1 & h2), which is average given that it needs to sort based on the heuristics of course.

A Search:* The *Optimally Efficient* search algorithm, A*, had one of the best performances overall. The memory usage ranged from 7~8MB (and spiked once to 10) which is also generally below average. The number of expanded nodes was not the lowest, but much lower than other optimal strategies like UCS, ranging between 11~130 (vs UC 13~554). CPU Usage had the range of 3.97~7.18% which can also be considered low compared to others. The A* showed generally a good balance between solution optimality, memory usage and cpu utilization.

Class Diagram: In the class diagram here, the search classes extend from the GeneralSearch class by having arrows pointing to the small box labeled GeneralSearch, which is the same as the General Search Class (the big box at the left).


WaterSortSearch
+ SHOW_DEBUG : boolean
+ heuristicVersion : int
+ currentAgent : GeneralSearch
+ BOTTLES_COUNT : int
+ MAX_BOTTLE_CAPACITY : int
+ solve(initialState : String, strategy : String, visualize : boolean) : String
- init(initialState : String, strategy : String, visualize : boolean) : void
+ EVAL_Fn(node : Node, version : int) : int
Node
- pathCost : int
- depth : int {readOnly}
- parent : Node
- state : State {readOnly}
+ pour : Action
+ Node(state : State, parent : Node, pour : Action, depth : int, pathCost : int)
+ Node(parent : Node)
+ Node(state : State)
+ applyAction(action : Action) : int
+ isGoal() : boolean
+ equals(obj : Object) : boolean
+ equals(node : Node) : boolean
+ hashCode() : int
+ toString() : String
+ getState() : State
+ getParent() : Node
+ getDepth() : int
+ getPathCost() : int
+ compareTo(o : Node) : int
+ getSolutionString(node : Node) : String
+ main(args : String[]) : void
boolean
Comparable<Node>
GeneralSearch
MaxDepth : int
+ expandedNodesCount : int
~ explored : HashSet<State>
~ initialState : State
~ solutionNode : Node
~ nodes : PriorityQueue<Node>
+ GeneralSearch()
+ GENERAL_SEARCH() : boolean
+ SEARCH_NextStep() : boolean
+ EVAL_Fn(node : Node) : int
+ EVAL_Fn(node : Node, version : int) : int
+ EXPAND(node : Node) : ArrayList<Node>
+ isSolutionFound() : boolean
+ GoalTest(node : Node) : boolean
+ removeFront() : Node
+ getFront() : Node
+ addNode(node : Node) : void
+ addNodes(nodes : ArrayList<Node>) : void
+ getSolutionNode() : Node
+ getExpandedNodesCount() : int
# getAgent(strategy : String) : GeneralSearch
+ setInitialState(initialState : State) : void
State
Action
GreedyS
+ EVAL_Fn(node : Node, version : int) : int
+ EVAL_Fn(node : Node) : int
+ H_n1(node : Node) : int
+ H_n2(node : Node) : int
AStar
+ EVAL_Fn(node : Node, version : int) : int
UCS
+ EVAL_Fn(node : Node, version : int) : int
DFS
+ EVAL_Fn(node : Node, version : int) : int
IDS
+ GENERAL_SEARCH() : boolean
BFS
+ EVAL_Fn(node : Node, version : int) : int
Node
int
GeneralSearch
State
- bottles : ArrayList<Color>[]
- init(bottles : ArrayList<Color>[]) : void
+ State()
+ State(input : String)
+ State(state : State)
+ isGoal() : boolean
+ equals(obj : Object) : boolean
+ equals(state : State) : boolean
+ hashCode() : int
+ clone() : Object
+ toString() : String
+ applyAction(action : Action) : int
+ getBottles() : ArrayList<Color>[]
+ getBottle(index : int) : ArrayList<Color>
+ main(args : String[]) : void
Action
+ to : int
+ from : int
+ Action(from : int, to : int)
+ toString() : String
<<enumeration>> SearchStrategy
Greedy
AStar
IDS
UCS
BFS
DFS
PriorityQueue<Node>
ArrayList<Color>[]
Cloneable
<<enumeration>> Color
EMPTY
orange
yellow
blue
green
red
HashSet<State>