src/Algorithms/PartB_BestF.java

```java
packageimportimportimportimportimportimportimportimportimportimportimport/
**
 * Implements the Best-First Search algorithm, an informed search that
 * prioritizes nodes based on a heuristic function. This class utilizes a
 * priority queue to manage the frontier, where nodes are ordered based on
their
 * heuristic values towards the goal, which helps in exploring the most
 * promising paths first. Best-First Search is particularly useful in
scenarios
 * where an approximate path to the goal is preferred quickly.
 */publicclassPartB_BestF/**
     * Executes the Best-First Search from a start node to a goal node
within a
     * specified planet size.
     *
     * @param@param@param@returnpublicstaticBestF(Node start, Node goal,
intnewPriorityQueuenewHashSetnewHashMapnullwhileNodecurrent=ifcontinueifreturn
forifreturnnull/**
     * Prints the current state of the frontier, showing the nodes in the
order they
     * will be explored, sorted by their heuristic values, and then by
angle and
     * distance when heuristics are equal.
     *
     * @paramprivatestaticvoidprintFrontier(PriorityQueue<Node> frontier)
newNode0if0Stringresult="%.3f"",""[""]" Algorithms;

 General.Node;
 General.Utility;

 java.util.Arrays;
 java.util.Comparator;
 java.util.HashMap;
 java.util.HashSet;
 java.util.List;
 java.util.Map;
 java.util.PriorityQueue;
 java.util.Set;
 java.util.stream.Collectors;


   {
     start      The starting node of the path.
     *  goal       The target node to reach.
     *  planetSize The size of the planet which may limit the search area.
     *  A list of nodes representing the path from start to goal if one
     *         exists; otherwise, returns null if no path can be found.
     */</span>
      List<Node>  planetSize)</span> {
       PriorityQueue<Node> frontier =  <>(
               Comparator.comparingDouble(Node::getHeuristic)
                       .thenComparingInt(Node::getAngle)
                       .thenComparingInt(Node::getD));
       Set<Node> visited =  <>();
       Map<Node, Node> parentMap =  <>();

       parentMap.put(start, );
       frontier.add(start);
```

```
50
51          (!frontier.isEmpty()) {
52             printFrontier(frontier);
53                frontier.poll();
54
55              (visited.contains(current)) {
56                 ;
57             }
58
59             visited.add(current);
60
61              (current.equals(goal)) {
62                List<Node> path = Utility.constructPath(current,
parentMap);
63                Utility.printPath(path, visited.size());
64                 path;
65             }
66             List<Node> successors = current.getSuccessors(planetSize,
goal);
67
68              (Node next : successors) {
69                  (!visited.contains(next) && !frontier.contains(next)) {
70                    frontier.add(next);
71                    parentMap.put(next, current);
72                 }
73             }
74          }
75          Utility.algorithmFails(visited.size());
76           ;
77      }
78
79       frontier The priority queue representing the nodes currently in the
80       *                 frontier of the search.
81       */</span>
82          {
83          Node[] frontierArray = frontier.toArray( []);
84          Arrays.sort(frontierArray,
85                  Comparator.comparingDouble(Node::getHeuristic)
86                          .thenComparingInt(Node::getAngle)
87                          .thenComparingInt(Node::getD));
88           (frontierArray.length != ) {
89                Arrays.stream(frontierArray)
90                     .map(node -> node.toString() + String.format(,
node.getHeuristic()))
91                     .collect(Collectors.joining());
92             System.out.println( + result + );
93          }
94      }
95 }
96
```