src/Algorithms/PartB_AStar.java

```
packageimportimportimportimportimportimportimportimportimportimportimport/**
 * Implements the A* (A-Star) search algorithm to find the most efficient path
 * from a start node to a goal node. A* is a best-first search algorithm that
 * uses costs along with heuristics to estimate the most promising path to the
 * goal. This class uses a priority queue to keep the nodes prioritized by the
 * estimated cost to reach the goal, combining the actual cost from the start
 * and a heuristic estimated cost to the goal.
 */publicclassPartB_AStar/**
     * Executes the A* search algorithm to find the shortest path from a start node
     * to a goal node within a specified planet size, considering both path cost and
     * heuristic estimates.
     *
     * @param@param@param@returnpublicstaticAStar(Node start, Node goal, intnewPriorityQueuenewHashMapnewHashMapintvisitCount=0null0.0whileNodecurrent=ifcontinuetrueifreturnfordoublenewCost=ifdoublepriority=returnnull/**
     * Prints the current state of the frontier in the A* algorithm, where the
     * frontier is prioritized by f-cost, and for nodes with the same f-cost, by
     * angle and then distance.
     *
     * @paramprivatestaticvoidprintFrontier(PriorityQueue<Node> frontier) newNode0if0Stringresult="%.3f"",""[""]" Algorithms;

General.Node;

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


  {
    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 shortest 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::getfCost)
                       .thenComparingInt(Node::getD)
```

```
                        .thenComparingInt(Node::getAngle));
        Map<Node, Node> parentMap =  <>();
        Map<Node, Double> costSoFar =  <>();
           ;
        frontier.add(start);
        parentMap.put(start, );
        costSoFar.put(start, );

         (!frontier.isEmpty()) {
            printFrontier(frontier);
               frontier.poll();
            visitCount++;

             (current.getVisited()) {
                ;
            }

            current.setVisited();

             (current.equals(goal)) {
                List<Node> path = Utility.constructPath(current, parentMap);
                Utility.printPath(path, visitCount);
                 path;
            }

            List<Node> successors = current.getSuccessors(planetSize, goal);

             (Node next : successors) {
                   costSoFar.get(current) + next.getCost();
                 (!costSoFar.containsKey(next) || newCost < costSoFar.get(next)) {
                    costSoFar.put(next, newCost);
                       newCost + next.calculateHeuristic(goal);
                    next.setfCost(priority);
                    frontier.add(next);
                    parentMap.put(next, current);
                }
            }
        }
        Utility.algorithmFails(visitCount);
         ;
    }

     frontier The priority queue representing the frontier of the A*
     *                  search.
     */</span>
        {
        Node[] frontierArray = frontier.toArray( []);
        Arrays.sort(frontierArray,
                Comparator.comparingDouble(Node::getfCost)
                        .thenComparingInt(Node::getAngle)
                        .thenComparingInt(Node::getD));
         (frontierArray.length != ) {
               Arrays.stream(frontierArray)
                    .map(node -> node.toString() + String.format(, node.getfCost()))
                    .collect(Collectors.joining());
            System.out.println( + result + );
        }
    }
}
```