src/Algorithms/PartB_SMAStar.java

```
packageimportimportimportimportimportimportimportimportimportimportimport/**
 * Implements the Simplified Memory-Bounded A* (SMA*) search algorithm, which is
 * a variant of the A* search algorithm designed to handle limited memory. It
 * dynamically adjusts the search frontier to maintain the size within a
 * specified memory bound, using a heuristic to prioritize nodes and truncating
 * the least promising nodes when necessary.
 */publicclassPartB_SMAStar/**
     * Executes the SMA* search from a start node to a goal node, considering a
     * defined memory size limit.
     *
     * @param@param@param@param@returnpublicstaticsmaStar(Node start, Node goal, intintnewPriorityQueuenewHashMapnewHashMapintvisitedCount=0null0.0whileNodecurrent=if10000.0breakifreturnreturnnull/**
     * Updates the frontier by adding successors of the current node, handling the
     * memory size constraint by potentially truncating less promising nodes.
     *
     * @param@param@param@param@param@paramprivatestaticvoidupdateFrontier(PriorityQueue<Node> frontier, Node current, Node goal, intintnewArrayList0// Process the nodes to add or remove to/from frontier outside of the iteration // over successorsnewArrayListforif10000trueifnullfalse// Now modify the frontierif// private static void updateFrontier(PriorityQueue<Node> frontier, Node// current, Node goal, int planetSize,// int memorySize,// Map<Node, Node> parentMap) {// List<Node> successors;// if (current.getForgotten().size() == 0) {// successors = current.getSuccessors(planetSize, goal);// } else {// successors = current.getForgotten();// }// for (Node successor : successors) {// if (current.getForgotten().contains(successor)) {// current.getForgotten().remove(successor);// } else {// if (!successor.equals(goal) && successor.getDepth() >= memorySize) {// successor.setfCost(10000);// }// }// successor.setLeaf(true);// successor.getParent().setLeaf(false);// }// frontier.addAll(successors);// if (frontier.size() > memorySize) {// shrinkFrontier(frontier, parentMap, goal, memorySize);// }// }/**
     * Reduces the size of the frontier when it exceeds the memory limit, removing
     * the least promising nodes.
     *
     * @param@param@param@paramprivatestaticvoidshrinkFrontier(PriorityQueue<Node> frontier, Map<Node, Node> parentMap, Node goal,
             intwhileNodeworstNode=ifnullfalseNodeparent=ifnulldoubleminFCost=10000iftrue/**
     * Checks if a node's parent is present in the frontier and if this parent is
     * the worst in terms of cost among its ancestors.
     *
     * @param@param@returnprivatestaticbooleanexistsInFrontierWhereWorstParentInAncestors(Node worstNode, PriorityQueue<Node> frontier)forifreturntruereturnfalse/**
     * Computes and returns the list of ancestor nodes for a given node up to the
     * root of the search tree.
```

```
    *
    * @param@returnprivatestaticancestors(Node node)newArrayListwhilenullreturn/**
    * Identifies the node with the worst (highest) f-cost that is a leaf node in
    * the frontier.
    *
    * @param@returnprivatestaticgetWorstLeafNode(PriorityQueue<Node> frontier)returnnull/**
    * Prints the nodes currently in the frontier along with their f-costs, sorted
    * by f-cost, angle, and distance.
    *
    * @paramprivatestaticvoidprintFrontier(PriorityQueue<Node> frontier) newNode0if0Stringresult="%.3f"","""[""]"/**
    * Constructs the path from the goal node back to the start node using the
    * parent map.
    *
    * @param@param@returnprivatestaticconstructPath(Node goal, Map<Node, Node> parentMap)newArrayListNodecurrent=whilenullreturn Algorithms;

General.Node;
General.Utility;

java.util.ArrayList;
java.util.Arrays;
java.util.Collections;
java.util.Comparator;
java.util.HashMap;
java.util.List;
java.util.Map;
java.util.PriorityQueue;
java.util.stream.Collectors;


  {
    start      The starting node of the search.
    *  goal       The goal node to find.
    *  planetSize The size of the planet, which may influence the maximum
    *                   search bounds.
    *  memorySize The maximum size of the frontier, limiting the number of
    *                   nodes stored in memory at one time.
    *  A list of nodes representing the path from the start to the goal if
    *         found; null if no path exists.
    */</span>
     List<Node>  planetSize,  memorySize)</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()) {
            visitedCount++;
            printFrontier(frontier);
               frontier.poll();

             (current.getfCost() >=  || current.getDepth() >= memorySize) {
                ;
            }

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

            updateFrontier(frontier, current, goal, planetSize, memorySize, parentMap);
        }
        Utility.algorithmFails(visitedCount);
         ;
    }

     frontier   The priority queue used to store nodes during the search.
     *  current    The current node being expanded.
     *  goal       The goal node of the search.
     *  planetSize The size of the planet influencing node expansions.
     *  memorySize The maximum number of nodes allowed in the frontier.
     *  parentMap  A map linking each node to its parent, used to reconstruct
     *                   paths.
     */</span>
        planetSize,
             memorySize, Map<Node, Node> parentMap)</span> {
        List<Node> successors =  <>(
                current.getForgotten().size() ==  ? current.getSuccessors(planetSize, goal) : current.getForgotten());



        List<Node> toAdd =  <>();
         (Node successor : successors) {
             (!successor.equals(goal) && successor.getDepth() >= memorySize) {
                successor.setfCost();
            }
            successor.setLeaf();
             (successor.getParent() != ) {
                successor.getParent().setLeaf();
            }
            toAdd.add(successor);
        }


        frontier.addAll(toAdd);

         (frontier.size() > memorySize) {
            shrinkFrontier(frontier, parentMap, goal, memorySize);
        }
    }

```

```
     frontier   The priority queue of nodes.
     *  parentMap  A map of nodes to their parents, used to maintain the
     *                   search tree's structure.
     *  goal       The goal node, used for recalculating heuristic values
if
     *                   needed.
     *  memorySize The maximum size of the frontier allowed.
     */</span>
        memorySize)</span> {
         (frontier.size() > memorySize) {
               getWorstLeafNode(frontier);
             (worstNode != ) {
                frontier.remove(worstNode);
                worstNode.setLeaf();
                worstNode.getForgotten().add(worstNode.getParent());
                   worstNode.getParent();
                 (parent != ) {
                    parent.getForgotten().add(worstNode);
parent.getForgotten().stream().mapToDouble(Node::getfCost).min().orElse();
                    parent.setfCost(minFCost);
                     (!
existsInFrontierWhereWorstParentInAncestors(worstNode, frontier)) {
                        parent.setLeaf();
                        frontier.add(parent);
                    }

                }
            }
        }
    }
```

```
 worstNode The node considered the worst based on its f-cost.
 *  frontier  The current frontier.
 *  true if the worst parent is found among the ancestors of any node in
 *         the frontier, false otherwise.
 */</span>
    {
     (Node node : frontier) {
         (ancestors(node).contains(worstNode.getParent())) {
             ;
        }
    }
     ;
}

 node The node whose ancestors are to be found.
 *  A list of nodes representing the ancestors of the given node.
 */</span>
  List<Node>  {
    List<Node> ancestors =  <Node>();
     (node.getParent() != ) {
        ancestors.add(node.getParent());
        node = node.getParent();
    }
     ancestors;
}

 frontier The current frontier.
 *  The node with the highest f-cost that does not have any children in
 *         the search tree.
 */</span>
  Node  {
     frontier.stream()
            .filter(node -> node.getLeaf())
            .max(Comparator.comparingDouble(Node::getfCost))
            .orElse();
}

 frontier The priority queue containing the nodes.
 */</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 + );
    }
}

 goal      The goal node where the path ends.
 *  parentMap A map of child nodes to their parent nodes as discovered
 *                  during the search.
 *  A list of nodes representing the path from the goal to the start.
 */</span>
```

```
      List<Node>  {
        List<Node> path =  <>();
           goal;
         (current != ) {
            path.add(current);
            current = current.getParent();
        }
        Collections.reverse(path);
         path;
    }
}
```