src/General/Node.java

```
  1 packageimportimportimport/**
  2  * Represents a node in a graph, typically used in pathfinding or graph
  3  * traversal algorithms. Each node is characterized by its position, cost
to
  4  * reach, heuristic estimate towards a goal, and various other attributes
useful
  5  * in search algorithms.
  6  */publicclassNodeimplementsComparableint// Distance from the poleint//
Angle in degrees// Parent node in the pathdouble// Cost to reach this node
double// Heuristic value of the node to the goalprivatedouble// f-cost: total
cost of the nodeint// Depth in the search treeboolean// Whether the node is
currently a leaf in the search tree// Nodes that were pruned from the search
frontierboolean/**
  7      * Constructs a new Node with specified parameters, used in SMA*
algorithm.
  8      *
  9      * @param@param@param@param@param@parampublicNode(intintdoubleintthis
thisthisthisifnullthiselsethis0thisthisthisthisthisfalsethisnewArrayListthis
false/**
 10      * Secondary constructor for Node without specifying depth, typically
used
 11      * outside the SMA* algorithm.
 12      *
 13      * @param@param@param@param@parampublicNode(intintdoublethisthisthis
thisifnullthiselsethis0thisthisthisthis1thisnewArrayListthisfalse/**
 14      * Calculates the heuristic estimate from this node to a specified
goal node
 15      * using the Euclidean distance formula adapted for polar coordinates.
 16      *
 17      * @param@returnpublicdoublecalculateHeuristic(Node goal)double
radCurrent=thisdoubleradGoal=returnthisthis2this/**
 18      * Generates successors for this node based on possible movements on
a grid.
 19      *
 20      * @param@param@returnpublicgetSuccessors(intnewArrayListint4545int11
forintintnewAngle=this360360ifthis0thisdoubleadditionalCost=thisnewNodethis
thisthisthis1forintintnewD=thisif0doubleadditionalCost=newNodethisthisthisthis
1return/**
 21      * Calculates the angular movement cost based on the change in angle
and the
 22      * current distance from the pole.
 23      *
 24      * @param@param@returnprivatedoublecalculateAngularCost(intdouble
doubleoneEighthCircumference=28return/**
 25      * Calculates the radial movement cost based on the change in
distance from the
 26      * pole.
 27      *
 28      * @param@returnprivatedoublecalculateRadialCost(intreturn/**
 29      * Calculates the distance from this node to another node,
considering both
 30      * angular and radial differences.
 31      *
 32      * @param@returnpublicdoubledistance(Node other)doubleangleChange=this
intradialChange=thisdoubleradialCost=thisdoubleangleCost=thisreturn/**
 33      * Compares this node to another node to determine ordering.
 34      * Nodes are first compared by distance from the pole, then by angle.
 35      *
```

```
 36      * @param@return@Overridepublicintcompare To(Node other)ifthisreturn
thisreturnthis/**
 37      * Determines whether another object is equal to this node.
 38      * Two nodes are considered equal if they have the same distance and
angle.
 39      *
 40      * @param@return@Overridepublicbooleanequals(Object obj)ifthisreturn
trueifnullreturnfalseNodeother=return/**
 41      * Generates a hash code for this node.
 42      *
 43      * @return@Overridepublicinthash Code()return/**
 44      * Returns a string representation of this node, typically used for
debugging
 45      * purposes.
 46      *
 47      * @return@OverridepublictoString()return"(%d:%d)"/**
 48      * Getters and Setters for the Node Class
 49      */publicdoublegetCost()returnpublicvoidsetCost(doublethispublic
getParent()returnpublicintgetD()returnpublicintgetAngle()returnpublicdouble
getHeuristic()returnpublicvoidsetHeuristic(doublethispublicdoublegetfCost()
returnpublicvoidsetfCost(doublethispublicgetForgotten()returnpublicvoid
setForgotten(List<Node> forgotten)thispublicintgetDepth()returnpublicvoid
setDepth(intthispublicbooleangetLeaf()returnpublicvoidsetLeaf(booleanthis
publicvoidsetVisited(booleanthispublicbooleangetVisited()return General;
 50  java.util.ArrayList;
 51  java.util.List;
 52  java.util.Objects;
 53
 54
 55     <Node> {
 56      d;
 57      angle;
 58     Node parent;
 59      cost;
 60      heuristic;
 61       fCost;
 62      depth;
 63      leaf;
 64     List<Node> forgotten;
 65      visited;
 66
 67      d      Distance from the pole.
 68      *  angle  Angle in degrees.
 69      *  parent Parent node in the search path.
 70      *  cost   Cost to reach this node.
 71      *  goal   Goal node to calculate heuristic towards.
 72      *  depth  Depth of this node in the search tree.
 73      */</span>
 74       d,  angle, Node parent,  cost, Node goal,  depth)</span> {
 75         .d = d;
 76         .angle = angle;
 77         .parent = parent;
 78         .cost = cost;
 79          (goal != ) {
 80             .heuristic = calculateHeuristic(goal);
 81         }  {
 82             .heuristic = ;
 83         }
 84         .fCost = .cost + .heuristic;
 85         .depth = depth;
 86         .leaf = ;
```

```
        .forgotten =  <Node>();
        .visited = ;
    }

     d      Distance from the pole.
     *  angle  Angle in degrees.
     *  parent Parent node.
     *  cost   Cost to reach this node.
     *  goal   Goal node to calculate heuristic towards.
     */</span>
      d,  angle, Node parent,  cost, Node goal)</span> {
        .d = d;
        .angle = angle;
        .parent = parent;
        .cost = cost;
         (goal != ) {
            .heuristic = calculateHeuristic(goal);
        }  {
            .heuristic = ;
        }
        .fCost = .cost + .heuristic;
        .depth = ;
        .forgotten =  <Node>();
        .visited = ;
    }

     goal The goal node to calculate the heuristic towards.
     *  The heuristic estimate.
     */</span>
       {
           Math.toRadians(.angle);
           Math.toRadians(goal.angle);
          Math.sqrt(.d * .d + goal.d * goal.d
                -  * .d * goal.d * Math.cos(radGoal - radCurrent));
    }

     planetSize The size of the planet, limiting the maximum allowable
     *                   distance.
     *  goal       The goal node, used to calculate heuristics for
successors.
     *  A list of successor nodes.
     */</span>
     List<Node>  planetSize, Node goal)</span> {
        List<Node> successors =  <>();
        [] angleChanges = { -,  };
        [] distanceChanges = { -,  };

         ( angleChange : angleChanges) {
              (.angle + angleChange + ) % ;
             (.d >  && .d < planetSize) {
                  calculateAngularCost(.d, angleChange);
                successors.add( (.d, newAngle, , .cost + additionalCost,
goal, .depth + ));
             }
        }

         ( distanceChange : distanceChanges) {
              .d + distanceChange;
             (newD >  && newD < planetSize) {
                  calculateRadialCost(distanceChange);
                successors.add( (newD, .angle, , .cost + additionalCost,
```

```
goal, .depth + ));
146             }
147         }
148          successors;
149     }
150 
151      d           The current distance from the pole.
152      *  angleChange The change in angle, in degrees.
153      *  The cost associated with the angular movement.
154      */</span>
155        d,  angleChange)</span> {
156            ( * Math.PI * d) / ;
157          oneEighthCircumference;
158     }
159 
160      distanceChange The change in distance from the current position.
161      *  The cost associated with the radial movement.
162      */</span>
163        distanceChange)</span> {
164          Math.abs(distanceChange);
165     }
166 
167      other The node to which the distance is calculated.
168      *  The calculated distance.
169      */</span>
170        {
171            .angle - other.getAngle();
172            .d - other.d;
173            .calculateRadialCost(radialChange);
174            .calculateAngularCost(radialChange, angleChange);
175          Math.max(radialCost, angleCost);
176     }
177 
178      other The node to compare against.
179      *  -1, 0, or 1 as this node is less than, equal to, or greater than
the
180      *         specified node.
181      */</span>
182 
183        {
184          (.d != other.d) {
185              Integer.compare(.d, other.d);
186          }
187          Integer.compare(.angle, other.angle);
188     }
189 
190      obj The object to compare with this node.
191      *  true if the specified object is equal to this node; false
otherwise.
192      */</span>
193 
194        {
195          ( == obj)
196              ;
197          (obj ==  || getClass() != obj.getClass())
198              ;
199            (Node) obj;
200          d == other.d && angle == other.angle;
201     }
202 
203      A hash code value for this node.
```

```
     */</span>

       {
         Objects.hash(d, angle);
    }

     A string representation of this node, showing its distance and angle.
     */</span>

     String  {
         String.format(, d, angle);
    }



       {
         cost;
    }

       cost)</span> {
        .cost = cost;
    }

     Node  {
         parent;
    }

       {
         d;
    }

       {
         angle;
    }

       {
         heuristic;
    }

       heuristic)</span> {
        .heuristic = heuristic;
    }

       {
         fCost;
    }

       fCost)</span> {
        .fCost = fCost;
    }

     List<Node>  {
         forgotten;
    }

       {
        .forgotten = forgotten;
    }

       {
         depth;
```

```
    }

       depth)</span> {
        .depth = depth;
    }

       {
         leaf;
    }

       leaf)</span> {
        .leaf = leaf;
    }

       visited)</span> {
        .visited = visited;
    }

      {
         visited;
    }
}

```