```java
package de.dakror.wseminar.graph.algorithm;

import static de.dakror.wseminar.util.Benchmark.Type.*;

import java.util.ArrayList;
import java.util.TreeSet;

import de.dakror.wseminar.Const.State;
import de.dakror.wseminar.graph.Edge;
import de.dakror.wseminar.graph.Graph;
import de.dakror.wseminar.graph.Path;
import de.dakror.wseminar.graph.Vertex;
import de.dakror.wseminar.graph.VertexData.Heuristics;
import de.dakror.wseminar.graph.VertexData.Position;
import de.dakror.wseminar.graph.WeightedEdge;
import de.dakror.wseminar.graph.algorithm.base.PathFinder;
import de.dakror.wseminar.util.Visualizer;

/**
 * @author Maximilian Stark | Dakror
 */
public class AStar<V> extends PathFinder<V> {
  TreeSet<Vertex<V>> openList;
  ArrayList<Vertex<V>> closedList;

  public AStar(Graph<Vertex<V>> graph, boolean animate) {
    super(graph, animate);
    openList = new TreeSet<>((a, b) -> Float.compare(a.get(Heuristics.class).F(),
    b.get(Heuristics.class).F()));
    closedList = new ArrayList<>();

    metaClasses = new Class<?>[] { Heuristics.class };
  }

  @SuppressWarnings("unchecked")
  @Override
  public Path<Vertex<V>> findPath(Vertex<V> from, Vertex<V> to) {
    Visualizer.resetAll(graph, true, false);
    BM.time();

    Heuristics<V> h = new Heuristics<>();
```

```
    h.H = distance(from, to);
    from.add(h);

    openList.add(from);
    BM.add(OPEN_LIST_SIZE);

    Vertex<V> last = null;
    while (true) {
      if (openList.size() == 0) return null;

      Vertex<V> v = openList.pollFirst();
      BM.add(SORTS);
      BM.sub(OPEN_LIST_SIZE);

      closedList.add(v);
      BM.add(CLOSED_LIST_SIZE);

      if (v.get(Heuristics.class).parent != null) Visualizer.setEdgePath(graph.getEdge(v,
      v.get(Heuristics.class).parent), true, true);
      Visualizer.setVertexState(v, State.CLOSEDLIST, true);

      if (takeStep(last, v, to)) break;

      last = v;
    }

    Path<Vertex<V>> p = new Path<Vertex<V>>();
    p.setUserData("AStar" + (animate ? " anim" : "") + " " + from.data() + "->" + to.data());
    Vertex<V> v = to;

    for (Edge<Vertex<V>> e : graph.getEdges()) {
      Visualizer.setEdgePath(e, false, true, false);
    }

    while (v != null) {
      p.add(0, v);
      if (v.get(Heuristics.class).parent != null) {
        Visualizer.setEdgePath(graph.getEdge(v, v.get(Heuristics.class).parent), true, true);
      }
      Visualizer.setVertexState(v, State.BACKTRACK, false);
      v = v.get(Heuristics.class).parent;

      BM.add(PATH_CREATION);
    }
    p.calculateCost(graph);

    p.setBenchmark(BM);

    BM.time();
    cleanup();
    Visualizer.resetAll(graph, true, false);

    return p;
  }

  @SuppressWarnings("unchecked")
  @Override
```

```
  protected boolean takeStep(Vertex<V> parent, Vertex<V> node, Vertex<V> to) {
    if (node.equals(to)) {
      to.add(node.get(Heuristics.class));
      return true;
    }

    float nG = node.get(Heuristics.class).G;
    for (Edge<Vertex<V>> e : graph.getEdgesFrom(node)) {
      Vertex<V> v = e.getOtherEnd(node);
      BM.add(v);
      if (v.get(Heuristics.class) == null) {
        Heuristics<V> h = new Heuristics<>();
        h.G = nG + weight(e);
        h.H = distance(v, to);
        h.parent = node;
        v.add(h);
        openList.add(v);
        Visualizer.setEdgeActive(e, true, true);
        Visualizer.setVertexState(v, State.OPENLIST, false);
        BM.add(OPEN_LIST_SIZE);
      } else if (nG + weight(e) < v.get(Heuristics.class).G) {
        BM.add(OVERRIDES);
        v.get(Heuristics.class).G = nG + weight(e);

        Visualizer.setEdgeActive(graph.getEdge(v.get(Heuristics.class).parent, v), false,
        true);

        v.get(Heuristics.class).parent = node;

        Visualizer.setEdgeActive(e, true, false);
      }
    }

    return false;
  }

  float weight(Edge<Vertex<V>> e) {
    return e instanceof WeightedEdge ? ((WeightedEdge<Vertex<V>>) e).getWeight() : 1;
  }

  float distance(Vertex<V> v, Vertex<V> to) {
    return v.get(Position.class).pos.dst(to.get(Position.class).pos);
  }
}
```