import java.util.HashMap;

import java.util.HashSet;

/\*\*

\* This class contains the implementation of the A\* pathfinding algorithm. The

\* algorithm is implemented as a static method, since the pathfinding algorithm

\* really doesn't need to maintain any state between invocations of the

\* algorithm.

\*/

public class AStarPathfinder

{

/\*\*

\* This constant holds a maximum cutoff limit for the cost of paths. If a

\* particular waypoint happens to exceed this cost limit, the waypoint is

\* discarded.

\*\*/

public static final float COST\_LIMIT = 1e6f;

/\*\*

\* Attempts to compute a path that navigates between the start and end

\* locations of the specified map. If a path can be found, the waypoint of

\* the <em>final</em> step in the path is returned; that waypoint can be

\* used to walk backwards to the starting point. If no path can be found,

\* <code>null</code> is returned.

\*\*/

public static Waypoint computePath(Map2D map)

{

// Variables necessary for the A\* search.

AStarState state = new AStarState(map);

Location finishLoc = map.getFinish();

// Set up a starting waypoint to kick off the A\* search.

Waypoint start = new Waypoint(map.getStart(), null);

start.setCosts(0, estimateTravelCost(start.getLocation(), finishLoc));

state.addOpenWaypoint(start);

Waypoint finalWaypoint = null;

boolean foundPath = false;

while (!foundPath && state.numOpenWaypoints() > 0)

{

// Find the "best" (i.e. lowest-cost) waypoint so far.

Waypoint best = state.getMinOpenWaypoint();

// If the best location is the finish location then we're done!

if (best.getLocation().equals(finishLoc))

{

finalWaypoint = best;

foundPath = true;

}

// Add/update all neighbors of the current best location. This is

// equivalent to trying all "next steps" from this location.

takeNextStep(best, state);

// Finally, move this location from the "open" list to the "closed"

// list.

state.closeWaypoint(best.getLocation());

}

return finalWaypoint;

}

/\*\*

\* This static helper method takes a waypoint, and generates all valid "next

\* steps" from that waypoint. The new waypoints are added to the "open

\* waypoints" collection of the passed-in A\* state object.

\*\*/

private static void takeNextStep(Waypoint currWP, AStarState state)

{

Location loc = currWP.getLocation();

Map2D map = state.getMap();

for (int y = loc.yCoord - 1; y <= loc.yCoord + 1; y++)

{

for (int x = loc.xCoord - 1; x <= loc.xCoord + 1; x++)

{

Location nextLoc = new Location(x, y);

// If "next location" is outside the map, skip it.

if (!map.contains(nextLoc))

continue;

// If "next location" is this location, skip it.

if (nextLoc == loc)

continue;

// If this location happens to already be in the "closed" set

// then continue on with the next location.

if (state.isLocationClosed(nextLoc))

continue;

// Make a waypoint for this "next location."

Waypoint nextWP = new Waypoint(nextLoc, currWP);

// OK, we cheat and use the cost estimate to compute the actual

// cost from the previous cell. Then, we add in the cost from

// the map cell we step onto, to incorporate barriers etc.

float prevCost = currWP.getPreviousCost() +

estimateTravelCost(currWP.getLocation(),

nextWP.getLocation());

prevCost += map.getCellValue(nextLoc);

// Skip this "next location" if it is too costly.

if (prevCost >= COST\_LIMIT)

continue;

nextWP.setCosts(prevCost,

estimateTravelCost(nextLoc, map.getFinish()));

// Add the waypoint to the set of open waypoints. If there

// happens to already be a waypoint for this location, the new

// waypoint only replaces the old waypoint if it is less costly

// than the old one.

state.addOpenWaypoint(nextWP);

}

}

}

/\*\*

\* Estimates the cost of traveling between the two specified locations.

\* The actual cost computed is just the straight-line distance between the

\* two locations.

\*\*/

private static float estimateTravelCost(Location currLoc, Location destLoc)

{

int dx = destLoc.xCoord - currLoc.xCoord;

int dy = destLoc.yCoord - currLoc.yCoord;

return (float) Math.sqrt(dx \* dx + dy \* dy);

}

}