Daily Coding Problem #87

Problem

This problem was asked by Uber.

A rule looks like this:

```
A NE B
```

This means this means point `A` is located northeast of point `B`.

```
A SW C
```

means that point `A` is southwest of `C`.

Given a list of rules, check if the sum of the rules validate. For example:

```
A N B
B NE C
C N A
```

does not validate, since `A` cannot be both north and south of `C`.

```
A NW B
A N B
```

is considered valid.

Solution

First, let's break down what it means for a list of rules to be invalid. Consider the following list of rules:

```
A N B
B N A
```

The second rule is obviously invalid, since the first node already stated that `B` is north of `A`. We can also see that the following list is equivalent and also invalid:

```
A N B
A S B
```

So, we can see that two rules invalidate each other if they relate the same pair of points and are in opposite directions. However, two rules do *not* invalidate each other if they are in the same or direction or are orthogonal to each other:

```
A N B
A E B
```

In this case, we see that it is valid for A to be North of and East of B at the same time.

```
A N B
C N B
```

In this case, the relative position of `A` and `C` is ambiguous, other than that they are both north of `B`.

Let's take a look at another example, similar to the first provided example:

```
A N B
B N C
C N A
```

In this case, we see that `C` cannot be north of `A` because it is implied that `C` is south of `A` by the previous two rules. We could have re-written the first two rules into the following, so that the contradiction is obvious:

```
A N B (original)
B S A
B N C (original)
A N C (through B)
C S B
C S A (through B)
```

Then, it is obvious that `C N A` and `C S A` are contradictory. We will perform this expansion and check for contradiction in our algorithm.

Next, we need to figure out how to deal with the diagonal cardinal directions (e.g. `NE`, `NW`, `SW`, `SE`). Let's take a look at the case where there are two rules relating the same two points, and the directions are orthogonal (perpendicular) to each other:

```
A N B
A E B
```

We also notice that these two rules can be simplified into one: `A NE B`. Similarly, we can break down any diagonal direction into the two simple directions (`N`, `E`, `S`, `W`) that it is composed of.

Now, we can model the relationships between points as a graph. For each point in the graph, there will be a corresponding vertex in the graph. To represent the cardinal directions, each vertex will have a list of edges, once for each of the four directions. In our solution, we will use directed edges with the convention that an edge `fromVertex DIR toVertex` means `toVertex` is "`DIR` *of*" `fromVertex`. For example, the rule `A N B` will be parsed into an `N` edge from `B` pointing to `A`, meaning `A` is `North` *of* `B`.

When we add a new relationship, we should add a bi-directional edge between the two vertices -- one for the direction in the rule, and one for the opposite. For example, if the rule is `A N B`, we should add an `N` edge from `B` to `A`, and an `S` edge from `A` to `B`.

To add diagonal relationships, we simply parse the two directions into single directions, and treat them as two separate rules.

To validate an rule, we need to check if any existing edges conflict with the new edge(s) we are adding. We compute the relationships between all existing vertices and the new `toVertex`, and cache these within the graph.

Then, we simply check all the neighbors of the `fromVertex`, and return `false` if the neighbor's relationship with `toVertex` is contradictory to the new relationship (i.e. `N` vs `S`, `E` vs `W`).

When we add a new rule, we need to similarly add the relationship to all neighbors of the fromNode. For example, say `A` is already north of `B` (and `B` is already south of `A`). If we add the relationship `C` south of `B`, we also add the relationship `C` south of `A` (and `A` north of `C`). If we add the relationship `C` west of `B`, we also add the relationship `C` west of `A` (and `A` east of `C`). However, we do not add a relationship to the neighbors in the same direction as the new relationship, as mentioned in an example above.

Time complexity: `O(N * |V|) = O(N^2)`, where `N` is the number of rules.

Space complexity: `O(|V| + |E|) = O(|V| + |V|^2) = O(N^2)`, since we are creating a densely-connected graph.

```
class Solution {
```

```
public static void main(String[] args) {
  test1();
  test2();
  test3();
}

private static void test1() {
  String[] rules = {"A N B",
                    "C SE B",
                    "C N A"};
  System.out.println(validate(rules));
}

private static void test2() {
  String[] rules = {"A NW B",
                    "A N B"};
  System.out.println(validate(rules));
}

private static void test3() {
  String[] rules = {"A N B",
                    "C N B"};
  System.out.println(validate(rules));
}

static class Node {
  List<Set<Node>> edges = new ArrayList<>();
  char val;
  public Node(char val) {
    this.val = val;
    for (int i = 0; i < 4; i++)
```

```
      edges.add(new HashSet<>());
  }
}

public static final int N = 0;
public static final int E = 1;
public static final int S = 2;
public static final int W = 3;
public static final int[] DIRS = {N, E, S, W};
public static final Map<Character, Integer> charToDir = new HashMap<>();;
static {
  charToDir.put('N', N);
  charToDir.put('E', E);
  charToDir.put('S', S);
  charToDir.put('W', W);
}

public static boolean validate(String[] rules) {
  Map<Character, Node> map = new HashMap<>();

  for (String line : rules) {
    String[] rule = line.split(" ");
    System.out.println("Rule " + rule[0] + " " + rule[1] + " " + rule[2]);
    char fromVal = rule[2].charAt(0);
    char toVal = rule[0].charAt(0);

    if (!map.containsKey(fromVal)) {
      Node n = new Node(fromVal);
      map.put(fromVal, n);
    }
```

```
    if (!map.containsKey(toVal)) {
      Node n = new Node(toVal);
      map.put(toVal, n);
    }

    Node from = map.get(fromVal);
    Node to = map.get(toVal);

    /* Decompose diagonal (two-char) directions to single directions */
    for (char dirChar : rule[1].toCharArray()) {
      int dir = charToDir.get(dirChar);
      if (!isValid(map, from, to, dir))
        return false;
      addEdges(map, from, to, dir);
      System.out.println(from.edges.get(dir));
       System.out.println(to.edges.get(opposite(dir)));
    }

  }

  return true;
}

private static int opposite(int dir) {
  return (dir + 2) % 4;
}

private static boolean isValid(Map<Character, Node> map,
                               Node from,
                               Node to,
                               int newDir) {
```

```
  int oppositeDir = opposite(newDir);
  if (from.edges.get(oppositeDir).contains(to))
        return false;

  return true;
}

private static void addEdges(Map<Character, Node> map,
                             Node from,
                             Node to,
                             int newDir) {
  /* Get the direct opposite direction, e.g. S from N */
  int oppositeDir = opposite(newDir);

  /* Add the immediate edge between the nodes, using bi-directional edges. */
  from.edges.get(newDir).add(to);
  to.edges.get(oppositeDir).add(from);

  for (int dir : DIRS) {
    /* Relationships in the same direction are ambiguous.
       For example, if A is north of B, and we are adding
       C north of B, we cannot say C is north of A. */
    if (dir == newDir)
      continue;

    for (Node neighbor : from.edges.get(dir)) {
      /* No need to add self-edges */
      if (neighbor == to)
        continue;
      /* Add bi-directional edges */
      neighbor.edges.get(newDir).add(to);
```

```
            to.edges.get(oppositeDir).add(neighbor);
          }
        }
      }
    }
```