```
package player.playeragent;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import edu.neu.ccs.demeterf.demfgen.lib.List;
import edu.neu.ccs.demeterf.demfgen.lib.List.Fold;
import edu.neu.ccs.demeterf.demfgen.lib.Entry;
import edu.neu.ccs.demeterf.demfgen.lib.Map;
import edu.neu.ccs.evergreen.ir.Relation;
import edu.neu.ccs.satsolver.OutputI;
import edu.neu.ccs.satsolver.PolynomialI;
import edu.neu.ccs.satsolver.SATSolverUtil;
import gen.*;

public class Utils {
        /*********************************
         * Project 3 Functions
         *********************************/
        /*
         * returns the break-even price for the given relation
         */
        public static double getBreakEven(Relation rel)
        {
                double[] derPoly;
                double[] normPoly;

                //if there are ever all zeros or all ones, return 1
                if(rel.q(0) > 0 || rel.q(3) > 0)
                {
                        return 1;
                }

                //get the individual polynomials for each row bias
                double[] p0 = biasLookup(0, rel.q(0));
                double[] p1 = biasLookup(1, rel.q(1));
                double[] p2 = biasLookup(2, rel.q(2));
                double[] p3 = biasLookup(3, rel.q(3));

                //normalize the polynomials, and take the derivative of the result
                normPoly = normalizePolynomials(p0, p1, p2, p3);
                derPoly = derivativeOfPoly(normPoly);

                //find the maximum value of the polynomial
                return findMax(derPoly, normPoly);
        }

        /*solves the given polynomial for the provided max value
         polynomial is assumed to be in the form of:
         poly[0]x^3 + poly[1]x^2 + poly[2]x + poly[3]
        */
        public static double solvePolynomial(double max, double[] poly)
        {
                double total = 0;
                // solve c[0]*x^3
                total = poly[0] * (Math.pow(max, 3));
                // solve c[1]*x^2 and add to total
                total += poly[1] * (Math.pow(max, 2));
                // solve c[2]*x and add to total
                total += poly[2] * max;
                // add c[4] to total
                total += poly[3];

                return total;
        }

        /*
         * Finds the derivative of a polynomial.
         * expects polynomial in form:
         *    poly[0]x^3 + poly[1]x^2 + poly[2]x + poly[3]
         * returns polynomial in form:
         *    3poly[1]x^2 + 2poly[1]x + poly[2]
         */
        public static double[] derivativeOfPoly(double[] poly)
        {
                double[] derivative = new double[3];

                derivative[0] = 3 * poly[0];
                derivative[1] = 2 * poly[1];
                derivative[2] = poly[2];
```

```
        return derivative;
    }

    /*
     * Adds the terms of the 4 polynomials together to get one large polynomial
     * in the form of:
     *   poly[0]x^3 + poly[1]x^2 + poly[2]x + poly[3]
     */
    public static double[] normalizePolynomials(double[] p0, double[] p1, double[] p2, double[] p3)
    {
        double[] normalizedPoly = new double[4];

        //loop through all terms, add them together and store them
        for(int i = 0; i < p0.length; i++)
        {
            normalizedPoly[i] = p0[i] + p1[i] + p2[i] + p3[i];
        }

        return normalizedPoly;
    }

    /* given a row and the q value return the bias in expanded form with the
     * proper coefficent already in place. returns a polynomial of the
     * form:
     *          poly[0]x^3 + poly[1]x^2 + poly[2]x + poly[3]
     */
    public static double[] biasLookup(int row, int qval)
    {
        //array is in form [0]b^3 + [1]b^2 + [2]b + [3]
        double[] bias = new double[4];
        switch(row)
        {
        case 0:
            bias[0] = -1 * qval;
            bias[1] = 3 * qval;
            bias[2] = -3 * qval;
            bias[3] = 1 * qval;
            break;
        case 1:
            bias[0] = 1 * qval;
            bias[1] = -2 * qval;
            bias[2] = 1 * qval;
            bias[3] = 0;
            break;
        case 2:
            bias[0] = -1  * qval;
            bias[1] = 1 * qval;
            bias[2] = 0;
            bias[3] = 0;
            break;
        case 3:
        default:
            bias[0] = 1  * qval;
            bias[1] = 0;
            bias[2] = 0;
            bias[3] = 0;
            break;
        }
        return bias;
    }

    /*find the roots of the polynomial given in the form:
     *      poly[1]x^2 + poly[1]x + poly[2]
     *by way of the quadratic formula or linear equation
     *once the roots are found, solve for them, and test them
     *according to the boundaries
     */
    public static double findMax(double[] der, double[] norm)
    {
        double[] roots = new double[2];
        double root = 0;
        double solved1, solved2 = 0;

        //if the quadratic can be solved
        if(der[0] != 0)
        {
            //sqrt(b^2 - 4ac)
            double squareRoot = Math.sqrt(Math.pow(der[1], 2) - (4*der[0]*der[2]));

            // (-b + sqrt(b^2 - 4ac)) / 2a
```

```
                        roots[0] = ((-1 * der[1]) + squareRoot) / (2 * der[0]);
                        // (-b - sqrt(b^2 - 4ac)) / 2a
                        roots[1] = ((-1 * der[1]) - squareRoot) / (2 * der[0]);

                        //solve for both roots
                        solved1 = solvePolynomial(roots[0], norm);
                        solved2 = solvePolynomial(roots[1], norm);

                        //if the first one is out of range, return 2nd
                        if(!(solved1 >= 0 && solved1 <= 1))
                        {
                                return solved2;
                        }
                        //if the second is out of range, return first
                        else if(!(solved2 >= 0 && solved2 <= 1))
                        {
                                return solved1;
                        }
                        //if both in range, return max
                        else
                        {
                                return Math.max(solved1, solved2);
                        }
                }

                //if a=0, solve linearly
                else
                {
                        root = (root - der[2]) / der[1];
                        return solvePolynomial(root, norm);
                }
        }
        /*********************************
         * Project 4 Functions
         *********************************/

        public static Set<Integer> allRelations(RawMaterial rm)
        {
                List<Constraint> cons = rm.instance.cs;
                Set<Integer> set = new HashSet<Integer>();

                for(int i = 0; i < cons.length(); i++)
                {
                        set.add(cons.lookup(i).r.v);
                }

                return set;
        }

        public static Set<Pair> allFractions (RawMaterial rm)
        {
                Set<Integer> allRels = allRelations(rm);
                List<Constraint> cons = rm.instance.cs;
                Iterator<Integer> itter = allRels.iterator();
                Set<Pair> pairs = new HashSet<Pair>();
                double totalWeight = 0.0;

                while(itter.hasNext())
                {
                        int relNum = itter.next().intValue();
                        double currWeight = 0.0;

                        //find a more elegant solution to this
                        totalWeight = 0.0;

                        for(int j = 0; j < cons.length(); j++)
                        {
                                //accumulate the weight field
                                totalWeight += cons.lookup(j).w.v;

                                if(cons.lookup(j).r.v == relNum)
                                {
                                        currWeight += cons.lookup(j).w.v;
                                }
                        }

                        pairs.add(new Pair(relNum, (currWeight / totalWeight)));
                }

                return pairs;
        }
```

```
        public static boolean PickBestAssignment(gen.Pair<PolynomialI, Double> pos, gen.Pair<Polynomial
I, Double> neg)
        {
                double negBreakEven = neg.a.eval(neg.b.doubleValue());
                double posBreakEven = pos.a.eval(pos.b.doubleValue());

                return(posBreakEven >= negBreakEven);
        }

        public static gen.Pair<PolynomialI, Double> PolyAndMaxBias(RawMaterial rm)
        {
                OutputI outi = SATSolverUtil.calculateBias(new InputInitial(rm));

                return new gen.Pair<PolynomialI, Double>(outi.getPolynomial(), outi.getMaxBias());

        }

        /*********************************
         * Project 5 Functions
         *********************************/

        @SuppressWarnings("unchecked")
        public static List<List<Relation>> findImplication(List<Relation> relations)
        {
                List<List<Relation>> implications = List.create();
                int rellength = relations.length();

                for (int i = 0; i < rellength; i++) {

                        for (int j = 0; j < rellength; j++) {
                                Relation relation1 = relations.lookup(i);
                                Relation relation2 = relations.lookup(j);

                                if (bitwiseCompare(relation1.getRelationNumber(), relation2.getRelation
Number())) {
                                        List<Relation> temp = List.create();

                                        temp = temp.push(relation1);
                                        temp = temp.push(relation2);

                                        if (!implications.contains(temp)) {
                                                implications = implications.push(temp);
                                        }
                                }
                        }
                }

                return implications;
        }

        public static boolean bitwiseCompare(int relation1, int relation2)
    {
            if (((relation1 & relation2) == relation1) && relation1 != relation2) {
                return true;
            } else {
                return false;
            }
    }

    /** Extracts the list of relations from a derivative */
    public static List<Relation> extractRelations(Derivative der)
    {
        List<Relation> relations = List.create();
        List<TypeInstance> instances = der.type.instances;

        for (int i = 0; i < instances.length(); i++) {
                Relation rel = new Relation(3, instances.lookup(i).r.v);
                relations = relations.push(rel);
        }

        return relations;
    }
    /*
    public static List<Relation> deleteImpliedRelations(List<Relation> rels)
    {
        List<List<Relation>> impliedRels = findImplication(rels);
        //List<Relation> notImplied = List.create();

        for(int i = 0; i < impliedRels.length(); i++)
        {
```

```
            List<Relation> temp = impliedRels.lookup(i);
            Relation implied = temp.lookup(1);
            if(rels.contains(implied))
            {
                rels = rels.remove(implied);
            }
        }

        return rels;
    }*/

    /***********************************
     * Project 8 Functions
     ***********************************/
    //the following functions are borrowed in their entirety from
    // the administrator code, ComputeQualty.java
        /** Compute the quality of an assignment == satRatio(reduce(rm,a)) */
        public static Quality quality(RawMaterial rm, Assignment a){
            RawMaterial reduced =
                a.literals.fold(new List.Fold<Literal,RawMaterial>(){
                    public RawMaterial fold(Literal lit, RawMaterial r){ return reduce(r, lit); }
                },rm);
            return new Quality(satisfiedRatio(reduced));
        }


        /** Reduce a given RawMaterial based on a single Literal assignment */
        public static RawMaterial reduce(RawMaterial s, final Literal lit){
            return new RawMaterial(new RawMaterialInstance(s.instance.cs.map(new List.Map<Constraint,Co
nstraint>(){
                public Constraint map(Constraint c){
                    int i = c.vs.index(lit.var);
                    if(i < 0 || (c.r.v == 255))return c;
                    return new Constraint(c.w, new RelationNr(new Relation(3, c.r.v)
                                .reduce(i, lit.value.sign())), c.vs);
                }})));
        }
        /** Accumulation for the satisfaction ratio */
        static class R{
            double top, bot;
            R(double t, double b){ top = t; bot = b; }
            R add(int t, int b){ return new R(top+t, bot+b); }
            double result(){ return bot==0.0?1.0:top/(double)bot; }
        }
        /** Calculate the satifaction ratio of a (reduced) RawMaterial */
        public static double satisfiedRatio(RawMaterial rm){
            return rm.instance.cs.fold(new List.Fold<Constraint,R>(){
                public R fold(Constraint c, R r){ return r.add((c.r.v == 255)?c.w.v:0, c.w.v); }
            }, new R(0,0)).result();
        }

        /*******WEEK 9 UTILS*********/
        // finds implied relations and deletes them, returning a shortened list
        public static List<Relation> deleteImpliedRelations (List<Relation> rels) {

           // make a copy of the given list to return as the result
           List<Relation> result = rels;
           Relation implied;

           for (int i = 0; i < rels.length(); i++) {
               for (int j = 0; j < rels.length(); j++) {
                   // don't compare an item in the list to itself
                   if (i == j) {
                       continue;
                   }

                   implied = rels.lookup(j);
                   // if i implies j, remove j from the result list if it hasn't been already
                   if (bitwiseCompare(rels.lookup(i).getRelationNumber(), implied.getRelationNumber()))

                   {
                       if (result.contains(implied)) {
                           result = result.remove(implied);
                       }
                   }
               }
           }

           return result;
        }
```

```
                /******* Week 10 Utils ********/
        // Find the value of b that maximizes the fraction of satisfied constraints
        public static double getBMax(Derivative d) {
                SimplePolynomial lookahead = getLookahead(d);
                List<Entry<Double, Double>> points = getCriticalPoints(lookahead);
                Entry<Double, Double> maxPoint = points.foldl(
                                new Fold<Entry<Double, Double>, Entry<Double, Double>>() {
                                        public Entry<Double, Double> fold(Entry<Double, Double> pair, E
ntry<Double, Double> max) {
                                                return pair.val > max.val ? pair : max;
                                                }
                                        }, new Entry<Double, Double>(0.0, 0.0));
                return maxPoint.key;
        }

        // Find the critical points of the given polynomial
        static List<Entry<Double, Double>> getCriticalPoints(SimplePolynomial lookahead)
        {
                Map<Double, Double> points = Map.create();
                points = points.put(0.0, lookahead.solve(0));
                points = points.put(1.0, lookahead.solve(1));
                double[] bmax = lookahead.getDerivative().getZeros();
                if (bmax.length == 1) {
                        if (bmax[0] > 0 && bmax[0] < 1)
                                points = points.put(bmax[0], lookahead.solve(bmax[0]));
                        } else if (bmax.length == 2) {
                                if (bmax[0] > 0 && bmax[0] < 1)
                                        points = points.put(bmax[0], lookahead.solve(bmax[0]));
                                if (bmax[1] > 0 && bmax[1] < 1)
                                        points = points.put(bmax[1], lookahead.solve(bmax[1]));
                                }
                return points.toList();
                }
        // Find the lookahead polynomial for the given derivative
        static SimplePolynomial getLookahead(Derivative d) {
                List<gen.TypeInstance> rels = d.type.instances;
                int q0 = 0;
                int q1 = 0;
                int q2 = 0;
                int q3 = 0;
                for (int i = 0; i < rels.length(); i++) {
                        Relation rel = new Relation(3, rels.lookup(i).r.v);
                        q0 += rel.q(0);
                        q1 += rel.q(1);
                        q2 += rel.q(2);
                        q3 += rel.q(3);
                }
                return new SimplePolynomial(
                                q3 - q2 + q1 - q0,
                                q2 - (2 * q1) + (3 * q0),
                                q1 - (3 * q0),
                                q0);
        }
        public static double getBreakEven(Derivative d)
        {
                SimplePolynomial poly = getLookahead(d);
                List<Entry<Double, Double>> points = getCriticalPoints(poly);
                Double breakEven =  points.foldl(new Fold<Entry<Double, Double>, Double>(){
                        public Double fold(Entry<Double, Double> pair, Double max){
                                return Math.max(pair.val, max);
                                }
                        }, 0.0);
                return breakEven;
        }
}
```