```
package util;

import edu.neu.ccs.evergreen.ir.*;
import java.util.*;

import edu.neu.ccs.demeterf.demfgen.lib.List;
import gen.*;

import java.util.HashMap;

// Class for finding break-even point for relations of rank 3
public class BreakEven {

        public static class BreakEvenPair{
                public double val, bias;
                public BreakEvenPair(double v, double b){
                        val = v;
                        bias = b;
                }
        }

        /** Keeps track of already computed single relation break-even values. */
        private static HashMap<Integer, BreakEvenPair> computedTB = new HashMap<Integer, BreakEvenPair>
();

        // Exception for non-quadratic arguments in the quadratic equation function
        @SuppressWarnings("serial")
        public static class NonQuadraticException extends Exception{}

        // Exception for non-linear arguments to the linear root-finder
        @SuppressWarnings("serial")
        public static class NonlinearException extends Exception{}

        // Exception for relations not supported by this class
        @SuppressWarnings("serial")
        public static class InvalidRelationException extends Exception{}

        // Class that stores quadratic roots
        private static class QuadraticRoots{
                public Double plus = Double.NaN, minus = Double.NaN;
        }

        // Returns the polynomial associated with q(3)
        private static Polynomial ThreeTrueNoneFalse(){
                LinkedList<Double> coeffs = new LinkedList<Double>();
                coeffs.add(0.0);
                coeffs.add(0.0);
                coeffs.add(0.0);
                coeffs.add(3.0);
                return Polynomial.create(coeffs);
        }

        // Returns the polynomial associated with q(2)
        private static Polynomial TwoTrueOneFalse(){
                LinkedList<Double> coeffs = new LinkedList<Double>();
                coeffs.add(0.0);
                coeffs.add(0.0);
                coeffs.add(1.0);
                coeffs.add(-1.0);
                return Polynomial.create(coeffs);
        }

        // Returns the polynomial associated with q(1)
        private static Polynomial OneTrueTwoFalse(){
                LinkedList<Double> coeffs = new LinkedList<Double>();
                coeffs.add(0.0);
                coeffs.add(1.0);
                coeffs.add(-2.0);
                coeffs.add(1.0);
                return Polynomial.create(coeffs);
        }

        // Returns the polynomial associated with q(0)
        private static Polynomial NoneTrueThreeFalse(){
                LinkedList<Double> coeffs = new LinkedList<Double>();
                coeffs.add(1.0);
                coeffs.add(-3.0);
                coeffs.add(3.0);
                coeffs.add(-1.0);
                return Polynomial.create(coeffs);
        }
```

```
        private static List<RelationI> evergreenConversion(List<RelationNr> descriptor){
                java.util.Iterator<RelationNr> iter = descriptor.iterator();
                List<RelationI> out = List.create();
                while(iter.hasNext()){
                        out = out.append(new Relation(3, iter.next().v));
                }
                return out;
        }

        /** Gets the relations from a Derivatie */
        public static List<RelationNr> getRelations(Derivative d){
                Iterator<TypeInstance> iter = d.type.instances.iterator();
                List<RelationNr> rels = List.create();
                while(iter.hasNext()){
                        rels.append(iter.next().r);
                }
                return rels;
        }

        /** Gets the Break-Even point for relations with length <= 2*/
        public static double getBreakEvenPoint(List<RelationNr> descriptor) throws InvalidRelationExcep
tion, NonQuadraticException{
                descriptor = player.Util.pruneImplied(descriptor);
                List<RelationI> rels = BreakEven.evergreenConversion(descriptor);

                if(rels.length() > 2 || rels.length() <= 0)
                        return 1.0;
                else if(rels.length() == 1)
                        return getBreakEvenPointTB(rels.top());

                // V(b,t) = t*look-ahead-a(b)+(1-t)*look-ahead-b(b)
                // dV/dt = look-ahead-a(b) - look-ahead-b(b)
                //Polynomial laa, lab, dvdt, dvdb;
                double oneBE = BreakEven.getBreakEvenPointTB(rels.lookup(0));
                double twoBE = BreakEven.getBreakEvenPointTB(rels.lookup(1));
                return oneBE < twoBE ? oneBE : twoBE;
        }

        /** Gets the Break-Even point for relations with length <= 2*/
        public static double getBreakEvenBias(List<RelationNr> descriptor) throws InvalidRelationExcept
ion, NonQuadraticException{
                descriptor = player.Util.pruneImplied(descriptor);
                List<RelationI> rels = BreakEven.evergreenConversion(descriptor);

                if(rels.length() > 2 || rels.length() <= 0)
                        return 1.0;
                else if(rels.length() == 1)
                        return getBreakEvenBiasTB(rels.top());

                // V(b,t) = t*look-ahead-a(b)+(1-t)*look-ahead-b(b)
                // dV/dt = look-ahead-a(b) - look-ahead-b(b)
                //Polynomial laa, lab, dvdt, dvdb;
                double oneBE = BreakEven.getBreakEvenPointTB(rels.lookup(0));
                double twoBE = BreakEven.getBreakEvenPointTB(rels.lookup(1));
                return oneBE < twoBE ? BreakEven.getBreakEvenBiasTB(rels.lookup(0)) : BreakEven.getBrea
kEvenBiasTB(rels.lookup(1));
        }

        /** Gets the BreakEvenPair (bias and break-even value) for the given TB relation. */
        private static BreakEvenPair getBreakEvenPairTB(RelationI rel)throws InvalidRelationException{
                if(rel.getRank() != 3)  // If the relation isn't or rank three this class/function does
 not support it.
                        throw new InvalidRelationException();
                if(computedTB.containsKey(rel.getRelationNumber())){
                        return computedTB.get(rel.getRelationNumber());
                }
                else{
                        BreakEvenPair temp = BreakEven.computeBreakEvenPairTB(rel);
                        computedTB.put(rel.getRelationNumber(), temp);
                        return temp;
                }
        }

        /** Computes the BreakEvenPair (bias and break-even value) for the given single relation. */
        private static BreakEvenPair computeBreakEvenPairTB(RelationI rel){
                if(rel.getRelationNumber() == 0)                                        // If the relat
ion number equals zero the relation can never be satisfied.
                        return new BreakEvenPair(0.0, 0.5);
                if(rel.q(0) > 0)
// If q(0) > 1 then this can be maximized at 1 by setting all variables to false.
```

```
                        return new BreakEvenPair(1.0, 0.0);
                if(rel.q(3) > 0)
// If q(3) > 1 then this can be maximized at 1 by setting all variables to true.
                        return new BreakEvenPair(1.0, 1.0);
                Polynomial biasedProb = BreakEven.getRelationProbabilityPolynomial(rel);
                Polynomial probDeriv = biasedProb.derivative();
                double evalOne = biasedProb.eval(1.0), evalZero = biasedProb.eval(0.0);
                BreakEvenPair oneZeroMax = (evalOne > evalZero) ? new BreakEvenPair(evalOne, 1.0) : new
 BreakEvenPair(evalZero, 0.0);//, xValMax = (evalOne > evalZero) ? 1.0 : 0.0;
                QuadraticRoots primeRoots;
                if(probDeriv.getOrder() == 2){
                        try{
                                primeRoots = BreakEven.applyQuadraticFormula(probDeriv);
                        }catch(NonQuadraticException e){
                                return oneZeroMax;
                        }
                }else if(probDeriv.getOrder() == 1){
                        try{
                                primeRoots = new QuadraticRoots();
                                primeRoots.minus = BreakEven.getLinearRoot(probDeriv);
                        }catch(NonlinearException e){
                                return oneZeroMax;
                        }
                }else{
                        return oneZeroMax;
                }
                if(!primeRoots.minus.equals(Double.NaN)){
                        if(primeRoots.minus > 0.0 && primeRoots.minus < 1.0){   // If on the interval [
0,1]
                                double eval = biasedProb.eval(primeRoots.minus);// Evaluate
                                if(eval > oneZeroMax.val)
        // If larger than the running max, then set the running max to the value evaluated.
                                        oneZeroMax = new BreakEvenPair(eval, primeRoots.minus);
                        }
                }
                if(!primeRoots.plus.equals(Double.NaN)){
                        if(primeRoots.plus > 0.0 && primeRoots.plus < 1.0){             // If on the in
terval [0,1]
                                double eval = biasedProb.eval(primeRoots.plus); // Evaluate
                                if(eval > oneZeroMax.val)
        // If larger than the running max, then set the running max to the value evaluated.
                                        oneZeroMax = new BreakEvenPair(eval, primeRoots.plus);
                        }
                }
                return oneZeroMax;      // return
        }

        // Gets the break even point for a 3 relation
        private static double getBreakEvenPointTB(RelationI rel)throws InvalidRelationException{
                return BreakEven.getBreakEvenPairTB(rel).val;
        }

        // Gets the break even point for a 3 relation
        private static double getBreakEvenBiasTB(RelationI rel)throws InvalidRelationException{
                return BreakEven.getBreakEvenPairTB(rel).bias;
        }

        // Gets the biased probability polynomial for the given relation of rank 3.
        private static Polynomial getRelationProbabilityPolynomial(RelationI rel){
                return BreakEven.ThreeTrueNoneFalse().multiplyByScalar(rel.q(3)).add(
                                BreakEven.NoneTrueThreeFalse().multiplyByScalar(rel.q(0)).add(
                                                BreakEven.OneTrueTwoFalse().multiplyByScalar(rel.q(1)).
add(
                                                                BreakEven.TwoTrueOneFalse().multiplyByS
calar(rel.q(2)))));
        }



    /** Takes the weighted fractions of relations mapped to those relations and returns the Polynomial
for those relations. */
    public static Polynomial getWeightedFractsLookAhead(Map<RelationNr, Double> rels){
        Polynomial poly = Polynomial.create();
        Iterator<RelationNr> iter = rels.keySet().iterator();
        while(iter.hasNext()){
                RelationNr tempRelNr = iter.next();
                Relation tempRel = new Relation(2, tempRelNr.v);
                poly = poly.add(BreakEven.getRelationProbabilityPolynomial(tempRel).multiplyByScalar(re
ls.get(tempRelNr)));
        }
        return poly;
```

```
    }

    /** Takes the weighted fractions of relations mapped to those relations and returns the Polynomial
for those relations. */
    public static Polynomial getRawMaterialLookAhead(RawMaterial r){
        return BreakEven.getWeightedFractsLookAhead(player.Util.getWeightedFracts(r));
    }

        // Finds the roots for quadratic polynomial f.
        // If roots are imaginary, this returns a QuadraticRoots with Double.NaN set as plus and minus.
        // If f is not quadratic this throws a NonQuadraticException.
        private static QuadraticRoots applyQuadraticFormula(Polynomial f)throws NonQuadraticException{
                if(f.getOrder() != 2)
                        throw new NonQuadraticException();
                QuadraticRoots retVals = new QuadraticRoots();
                double a = f.getCoefficient(2), b = f.getCoefficient(1), c = f.getCoefficient(0);
                double bSqr = b*b, fourAC = 4.0*a*c;
                if(fourAC > bSqr)
                        return retVals; // Roots are imaginary.
                double leadingVal = (-b)/(2.0*a), addSubVal = Math.sqrt(bSqr-fourAC)/(2.0*a);
                retVals.plus = leadingVal + addSubVal;
                retVals.minus = leadingVal - addSubVal;
                return retVals;
        }

        // Finds the root of a linear polynomial
        private static double getLinearRoot(Polynomial f)throws NonlinearException{
                if(f.getOrder() != 1)
                        throw new NonlinearException();
                return -(f.getCoefficient(0)/f.getCoefficient(1));
        }
}
```