```java
import edu.mit.six825.bn.functiontable.*;
import edu.mit.six825.bn.bayesnet.*;
import edu.mit.six825.bn.inputs.*;
import techniques.utils.Random;

import java.util.*;

/**
 * An implementation of the Variable Elimination bayes-nets solver,
 * implemented as per the pseudo-code provided in Koller & Friedman [2005]
 * pp.226-235
 *
 * @author nocturne
 */
public class VESolver extends Solver {
    protected static ElimOrderer _elimOrderFunc = new DumbOrder();

    public void setOrderer(ElimOrderer eo) {
        this._elimOrderFunc = eo;
    }

    public String toString() {
        return "VESolver";
    }

    public static void main(String[] args) {

        BayesNet bn = Nets.getBurglary();

        Solver solver = new VESolver();
        solver.setBayesNet(bn);

        try {
            Assignment evidence =
                ConstructEvidence(bn.nodes,
                                  new String[] {"JohnCalls", "true",
                                                "MaryCalls", "true"});
            solver.setEvidence(evidence);
        }
        catch (IllegalArgumentException e) {
            System.out.println("ERROR: " + e);
            return;
        }

        // Have to explicitly cast solver to a VESolver in order to
        // access setOrderer() method!

        ((VESolver)solver).setOrderer(new GreedyOrder());
        //((VESolver)solver).setOrderer(new RandomOrder());
        //((VESolver)solver).setOrderer(new DumbOrder());

        Function answer = solver.query(bn.nodes.getNode("Burglary"));
        System.out.println(answer);

        System.out.println("-----------------");

        bn = Nets.getInsurance();

        solver.setBayesNet(bn);
        try {
            Assignment evidence =
                ConstructEvidence(bn.nodes,
                                  new String[] {"Age", "Adolescent",
                                                "Airbag", "False",
                                                "Mileage", "TwentyThou"});
            solver.setEvidence(evidence);
        }
        catch (IllegalArgumentException e) {
            System.out.println("ERROR: " + e);
            return;
        }
        answer = solver.query(bn.nodes.getNode("PropCost"));
        System.out.println(answer);
    }

    public Function query (BayesNetNode variable) {
        FunctionVariable [] queryvar = new FunctionVariable[] {variable.var};

        Assignment e = _evidence;
        if (e == null) {
            // if evidence is null, whip up a null assignment and
            // splice that in, to keep the rest of our code happy.
            e = new Assignment(new FunctionVariableSet(queryvar[0]));
            e = e.subtract(e); // I.e. e = <the empty assignment>
            System.out.println("Warning: performing evidenceless query.");
        }

        BayesNetNodeSet bnset = GrabRelevantNodes(_bn.nodes, queryvar, e);
        System.out.println("BN shrank from " + _bn.nodes.size() + " to " +
                           bnset.size());

        return (CondProbVE(bnset, queryvar, e));
    }

    public VESolver(BayesNet _bn) {
        super(_bn);
    }
    public VESolver() {
        super();
    }


    /*******************************************************************
     **                                                               **
     **                  VE Algorithm Subroutines                     **
     **                                                               **/

    /**
     * @return normalized results of querying query variables, in
     * bayesnet bn, given evidence
     */
    public static Function CondProbVE (BayesNetNodeSet bnodes,
                                       FunctionVariable [] queryvars,
                                       Assignment evidence) {

        // Make sure we're only looking at the relevant nodes.
        bnodes = GrabRelevantNodes(bnodes, queryvars, evidence);

        Function [] cpts = new Function[bnodes.size()];

        for (int i=0; i < bnodes.size(); i++) {
            BayesNetNode n = bnodes.getNode(i);
            // System.out.println(n + " --> " + n.parents);
            if (n.cpt.variables.isSubsetOf(evidence.variables)) {
                // This node is rendered irrelevant by the evidence.
                // We will have to filter this null out in a moment...
                cpts[i] = null;
            } else if (n.cpt.variables.containsAnyOf(evidence.variables)) {
                cpts[i] = FilterForEvidence(n.cpt, evidence);
            } else {
```

```java
            cpts[i] = n.cpt;
        }
    }
    // Strip out the null CPTs resulting from the cases where the
    // original CPT vars were a subset of the evidence vars
    cpts = RemoveNullElts(cpts);

    FunctionVariableSet toEliminate =
        VarsToElim(bnodes, queryvars, evidence);

    return Compute.normalize(SumProductVE(cpts,
                                          queryvars,
                                          toEliminate,
                                          _elimOrderFunc));
}

/**
 * @return product of factors remaining after elimination, with
 * variables elimvars eliminated in the order determined by the
 * ordering function ElimOrderer.
 */
public static Function SumProductVE(Function [] factors,
                                    FunctionVariable [] query,
                                    FunctionVariableSet toEliminate,
                                    ElimOrderer Orderer) {
    /* Implementation of algorithm on K&F p.228 */

    while (toEliminate.size() > 0) {
        FunctionVariable nextvar =
            Orderer.doNext(factors, toEliminate, query);

        toEliminate = toEliminate.subtract(nextvar);

        factors = SumProductEliminateVar(factors, nextvar);
    }

    return(FactorProduct(factors));
}

/**
 * @return factors, with variable elimvar eliminated
 */
public static Function[] SumProductEliminateVar(Function [] factors,
                                                FunctionVariable elimvar) {
    /* Implementation of algorithm on K&F p.228 */

    List AlongForTheRide = (List)new ArrayList();
    List Condensing      = (List)new ArrayList();

    for (int i=0; i < factors.length; i++) {
        if (factors[i].variables.contains(elimvar)) {
            Condensing.add(factors[i]);
        } else {
            AlongForTheRide.add(factors[i]);
        }
    }

    if (Condensing.isEmpty()) {
        String factstr = "";
        for (int i=0; i < factors.length; i++, factstr += ", ") {
            factstr += factors[i].variables.toString();
        }
        throw new IllegalArgumentException(
                   "Variable \"" + elimvar + "\" is not present in "
                   + "any of the given factors: " + factstr);
```

```java
    }

    Function product = FactorProduct(FuncCollToFArray(Condensing));

    Function tau = FactorMarginalize(product, elimvar);

    // It's possible we had some evidence variable that was off in
    // la la land, somewhere completely disconnected from the
    // query (and hence irrelevant to the query). My code doesn't
    // attempt to detect this in advance, so when the last
    // remaining variable from the disconnected/irrelevant island
    // is marginalized, you get an empty factor. That is caught
    // here: when this happens, tau will be empty
    if (tau.variables.size() > 0) {
        // Strictly speaking, AFtR is no longer just along for the ride
        AlongForTheRide.add(tau);
    } else {
        System.out.println("Some evidence variable was irrelevant "
                           + "to the query.");
    }
    return (FuncCollToFArray(AlongForTheRide));
}

/**                                                                 **
 **                                                                 **
 **                                                                 **
 ******************************************************************/

/******************************************************************
 **                                                                 **
 **                     CPT/Factor Utilities                        **
 **                                                                 **/

/**
 * @return factor product of the factors in array /funcs/
 */
public static Function FactorProduct(Function [] funcs) {
    if (funcs.length < 1) {
        throw new IllegalArgumentException("Cannot take null product");
    }
    // Start with first factor
    Function product = funcs[0];
    // Iterate over any remaining factors
    for (int i=1; i < funcs.length; i++) {
        product = FactorProduct(product, funcs[i]);
    }
    return (product);
}

/**
 * @return factor product of the CPTs Fa and Fb
 */
public static Function FactorProduct(Function Fa, Function Fb) {
    FunctionVariableSet FaVars = Fa.variables;
    FunctionVariableSet FbVars = Fb.variables;

    FunctionVariableSet ResultVars = FunctionVariableSet.union(FaVars,
                                                                 FbVars);

    if (ResultVars.size() == (FaVars.size() + FbVars.size())) {
        throw new IllegalArgumentException(
                    "Can/should not compute factor product of functions "
                    + "with no common variables: " + FaVars.toString()
                    + " x " + FbVars.toString());
    }
```

```java
        double [] entries = new double[ResultVars.cartesianProductSize()];

        System.out.println("Creating CPT of dimension " + ResultVars.size()
                           + " (" + entries.length + " entries)");
        /*
         * The structure of this Function-population mechanism was
         * largely stolen from the actual implementation of the
         * Function(FunctionVariableSet, double[]) constructor. The
         * expected form of that constructor's arguments is specified
         * only by the code (i.e. there's no documentation which
         * explains what's going on, so UTSL).
         */
        for (Iterator i = ResultVars.assignmentIterator(); i.hasNext(); ) {
            Assignment a = (Assignment) i.next();

            /* We could do index bookkeeping more efficiently here,
               but I have better things to worry about. */
            int index = a.computePosition();
            entries[index] = Fa.evaluate(a) * Fb.evaluate(a);
        }

        return new Function(ResultVars, entries);
    }

    /**
     * @return factor marginalization of /var/ in /cpt/ (sum out var from cpt)
     */
    public static Function FactorMarginalize(Function cpt,
                                             FunctionVariable var) {
        if (!cpt.variables.contains(var)) {
            throw new IllegalArgumentException(
                        "Cannot marginalize variable " + var.toString()
                        + " in a CPT which does not contain it: "
                        + cpt.variables.toString());
        }

        FunctionVariableSet ResultVars = cpt.variables.subtract(var);

        double [] entries = new double[ResultVars.cartesianProductSize()];

        /*
         * See comments in FactorProduct regarding use of the Function
         * constructor.
         */
        for (Iterator i = ResultVars.assignmentIterator(); i.hasNext(); ) {
            Assignment a = (Assignment) i.next();

            // sum over the domain of var
            double sum = 0;
            for (final Iterator j = var.domain.iterator(); j.hasNext(); ) {
                final DomainIndex dindex = (DomainIndex)j.next();
                final Comparable value = dindex.getValue();

                final Assignment subass = new Assignment(a, var, value);

                sum += cpt.evaluate(subass);
            }

            /* We could do index bookkeeping more efficiently here,
               but I have better things to worry about. */
            int index = a.computePosition();
            entries[index] = sum;
        }
        return new Function(ResultVars, entries);
    }

    /**                                                                   **
     **                                                                   **
     **                                                                   **
     *********************************************************************/

    /*********************************************************************
     **                                                                   **
     **                     Elimination Orderers                          **
     **                                                                   **/

    // An abstract interface which effectively lets us have function
    // pointers to elimination-ordering functions.
    //
    // Arguments: bnset -- must contain all bayes-net nodes relevant
    //                  to given query+evidence
    //               query -- array of query variables
    //               evidence -- assignment of evidence variables
    //
    // Returns: array of FunctionVariables in order of elimination
    //           (index 0 is first to be eliminated)
    //
    public static abstract interface ElimOrderer {
        public FunctionVariable doNext(Function [] factors,
                                       FunctionVariableSet eliminating,
                                       FunctionVariable [] query);
    }

    // An elimination-ordering instantiation which orders the
    // elimination variables randomly. (Nondeterministically: order
    // will be different on from invocation to invocation.)
    public static class RandomOrder implements ElimOrderer {
        public FunctionVariable doNext(Function [] factors,
                                       FunctionVariableSet eliminating,
                                       FunctionVariable [] query) {
            FunctionVariable fv =
                eliminating.getVariable(Random.random(eliminating.size()));
            return(fv);
        }
    }

    // A dumb orderer implementation which orders the variables in the
    // order they're given in the FVSet /eliminating/. (Which is
    // deterministically ordered for a given query method, so this
    // will generally have the same results given a particular query
    // formulation.)
    public static class DumbOrder implements ElimOrderer {
        public FunctionVariable doNext(Function [] factors,
                                         FunctionVariableSet eliminating,
                                         FunctionVariable [] query) {
            return(eliminating.getVariable(0));
        }
    }

    // A greedy elimination-ordering instantiation which picks the
    // variable whose elimination from the current set of factors
    // yields the smallest resulting factor.
    public static class GreedyOrder implements ElimOrderer {
        public FunctionVariable doNext(Function [] factors,
                                       FunctionVariableSet eliminating,
                                       FunctionVariable [] query) {
            // seed with bogus values that will be immediately overwritten
            int smallestsize = java.lang.Integer.MAX_VALUE;
```

```java
        FunctionVariable bestvar = eliminating.getVariable(0);

        for (int i=0; i < eliminating.size(); i++) {
            int size = WouldMakeFactorSize(eliminating.getVariable(i),
                                           factors);
            if (size < smallestsize) {
                smallestsize = size;
                bestvar = eliminating.getVariable(i);
            }
        }
        return(bestvar);
    }
}

/**                                                                  **
 **                                                                  **
 **                                                                  **
 *******************************************************************/

/*******************************************************************
 **                                                                  **
 **                  Random Utility Functions                        **
 **                                                                  **/

/**
 * @return the size (# of entries) of the CPT which would be
 * created by the elimination of var from the given factors.
 */
public static int WouldMakeFactorSize(FunctionVariable var,
                                      Function [] factors) {
    // Really, we want an empty FuncVarSet here, but FVS.union
    // assumes that no FVSet is empty (and will blow out if that's
    // not the case), and so we can't do that. So, instead,
    // initialize vars to contain just the query var, which we're
    // going to remove at the end anyway; this yields the desired
    // end result.
    FunctionVariableSet vars =
        new FunctionVariableSet(new FunctionVariable[] {var});

    for (int i=0; i < factors.length; i++) {
        if (factors[i].variables.contains(var)) {
            vars = FunctionVariableSet.union(vars,
                                             factors[i].variables);
        }
    }
    return(vars.subtract(var).cartesianProductSize());
}

/**
 * @return set of variables which will need to be eliminated from
 * bnodes in order to answer a query on queryvars (given evidence)
 */
public static FunctionVariableSet VarsToElim(BayesNetNodeSet bnodes,
                                             FunctionVariable[] queryvars,
                                             Assignment evidence) {
    FunctionVariableSet bnVars = bnodes.getFunctionVariableSet();
    FunctionVariableSet  qVars = new FunctionVariableSet(queryvars);
    FunctionVariableSet  eVars = evidence.variables;

    return(bnVars.subtract(qVars).subtract(eVars));
}


/**
 * @return nodeset of our query/evidence nodes plus those nodes
 * from bayesnet bn which are relevant to (i.e. an ancestor of)
 * the query/evidence nodes.
 */
public static BayesNetNodeSet GrabRelevantNodes(BayesNetNodeSet bnodes,
                                   FunctionVariable [] queryvars,
                                   Assignment evidence) {

    // Set of nodes we care about
    Set childNodes = (Set) new HashSet();

    // Feed query variables into child set
    for (int i=0; i < queryvars.length; i++) {
        childNodes.add(FindVariableNode(bnodes, queryvars[i]));
    }

    // And all the evidence variables
    FunctionVariableSet evars = evidence.variables;
    for (int i=0; i < evars.size(); i++) {
        childNodes.add(FindVariableNode(bnodes, evars.getVariable(i)));
    }

    BayesNetNodeSet childSet = NodeCollToBNNSet(childNodes);

    // And return the child nodes plus all their ancestors
    return (NodesAndAncestors(childSet));
}

public static BayesNetNodeSet NodesAndAncestors(BayesNetNodeSet nodes) {
    // Fringe of relevant nodes to be expanded in the BN graph
    List nodeFringe   = (List) new ArrayList();
    // Set of the previously-expanded nodes and/or ancestors
    Set familyNodes = (Set) new HashSet();

    // Initialize fringe to be all our input nodes
    for (int i=0; i < nodes.size(); i++) {
        nodeFringe.add(nodes.getNode(i));
    }

    // Process and expand the fringe until we're done
    while (nodeFringe.size() > 0) {
        // Grab some node from the fringe
        BayesNetNode n = (BayesNetNode) nodeFringe.remove(0);

        // And, if we haven't already looked at it...
        if (!familyNodes.contains(n)) {
            // ...add it to the relevant node set
            familyNodes.add(n);
            // ...and add its parents to the fringe
            for (int i=0; i< n.parents.size(); i++) {
                nodeFringe.add(n.parents.getNode(i));
            }
        }
    }

    return(NodeCollToBNNSet(familyNodes));
}

// Syntactic sugar for converting a Func collection to an array
public static Function[] FuncCollToFArray(Collection fcoll) {
    return((Function[]) fcoll.toArray(new Function[0]));
}

// Syntactic sugar for converting a BNN collection to a BNNSet
public static BayesNetNodeSet NodeCollToBNNSet(Collection ncoll) {
    BayesNetNode [] narray =
```

```java
            (BayesNetNode[]) ncoll.toArray(new BayesNetNode[0]);
        return(new BayesNetNodeSet(narray));
    }

    /**
     * @return BayesNet node for var (from network /bnodes/)
     */
    public static BayesNetNode FindVariableNode(BayesNetNodeSet bnodes,
                                                FunctionVariable var) {
        for (final Iterator i = bnodes.iterator(); i.hasNext(); ) {
            BayesNetNode n = (BayesNetNode) i.next();
            if (n.var.equals(var)) {
                return (n);
            }
        }
        throw new IllegalArgumentException(
                    "Could not find node for variable " + var);
    }

    /**
     * @return cpt|_(E=e)_ --- i.e. construct and return a filtered
     * CPT (based on the input CPT) which is consistent with the
     * evidence e. The evidence variables will not be in the scope of
     * the resulting CPT; the resulting scope will be the scope of the
     * input CPT minus the set of variables assigned by the evidence.
     *
     * This is used by Cond-Prob-VE line 2 (K&F p. 235).
     */
    public static Function FilterForEvidence(Function cpt, Assignment e) {
        FunctionVariableSet EVars      = e.variables;
        FunctionVariableSet ResultVars = cpt.variables.subtract(EVars);

        if (ResultVars.size() == 0) {
            throw new IllegalArgumentException(
               "FilterForEvidence should never be given cpt+evidence "
               + "where cpt vars are a subset of evidence vars");
        }

        // For efficiency's sake, remove from EVars the variables not
        // present in this particular CPT:
        FunctionVariableSet IgnorableVars = EVars.subtract(cpt.variables);
        EVars = EVars.subtract(IgnorableVars);

        double [] entries = new double[ResultVars.cartesianProductSize()];
        /*
         * See comments in FactorProduct regarding use of the Function
         * constructor.
         */
        for (Iterator i = ResultVars.assignmentIterator(); i.hasNext(); ) {
            /* We're iterating over all possible assignments of
               the *result* variables */
            Assignment ass = (Assignment) i.next();

            // fold the assignments from our evidence into this assignment
            for (int j=0; j < EVars.size(); j++) {
                FunctionVariable v = EVars.getVariable(j);
                ass = new Assignment(ass, v, e.getAssignedValue(v));
            }

            double wanted = cpt.evaluate(ass);

            /* compute the position only relative to the ResultVars
               (rather than all the vars in the assignment): only a subset
               of the assignment variables are actually indices in the
               function/CPT we're constructing. */
            /* (Note, Assignment.computePosition(FunctionVariableSet) is
               not really documented, but its usage elsewhere indicates
               the vars in the assignment must be a superset of the vars
               in the FunctionVariableSet.) */
            int index = ass.computePosition(ResultVars);
            entries[index] = wanted;
        }

        return new Function(ResultVars, entries);
    }

    /**
     * @return Comparable object corresponding to value with
     * stringification "val" in domain of FunctionVariable /var/.
     * (This lets us look up Comparable objects on the basis of their
     * string representation, which is kind of handy.)
     */
    public static Comparable FindVariableValueObj(FunctionVariable var,
                                                  String val) {
        Domain d = var.domain;
        for (int i=0; i < d.size(); i++) {
            if (d.getValue(i).toString().equals(val)) {
                return(d.getValue(i));
            }
        }
        throw new IllegalArgumentException(d + " does not include value \""
                                           + val + "\"!");
    }

    /**
     * @return Assignment corresponding to the evidence specified by
     * estrings, according to nodes in bn. Estrings must contain an
     * even number of strings: for any even /x/ estrings[x] is the
     * (stringified) name of a variable, and estrings[x+1] is the
     * (stringified) value to which that variable should be fixed.
     */
    public static Assignment ConstructEvidence(BayesNetNodeSet bn,
                                               String[] estrings) {
        if ((estrings.length % 2) != 0) {
            throw new IllegalArgumentException("ConstructEvidence given "
                         + "odd number of strings: " + estrings);
        }

        FunctionVariable [] eVars = new FunctionVariable[estrings.length/2];
        Comparable       [] eVals = new Comparable[estrings.length/2];

        for (int i=0; i < estrings.length; i += 2) {
            String nameString = estrings[i];
            String valString  = estrings[i+1];
            BayesNetNode n = bn.getNode(nameString);
            if (n == null) {
                throw new IllegalArgumentException("No such node \""
                                                   + nameString + "\"");
            }
            eVars[i/2] = n.var;
            eVals[i/2] = FindVariableValueObj(n.var, valString);
        }

        return(new Assignment(new FunctionVariableSet(eVars), eVals));
    }

    /**
     * @return array containing the non-null elements of arr.
     */
    public static Function[] RemoveNullElts(Function[] arr) {
```

```java
        int nulls = 0;
        for (int i=0; i<arr.length; i++) {
            if (arr[i] == null) nulls++;
        }
        if (nulls == 0) {
            return arr;
        } else {
            Function[] answer = new Function[arr.length - nulls];
            int j=0;
            for (int i=0; i<arr.length; i++) {
                if (arr[i] != null) {
                    answer[j++] = arr[i];
                }
            }
            return(answer);
        }
    }

}
```

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import edu.mit.six825.bn.functiontable.*;
import edu.mit.six825.bn.bayesnet.*;
import edu.mit.six825.bn.inputs.*;
import techniques.utils.*;
import java.util.Map;
import java.util.HashMap;
import java.util.ArrayList;

/**
 * An implementation of the Gibbs sampling algorithm for Bayes nets.
 *
 * @author drkp
 */
public class GibbsSolver extends Solver {
    private static final boolean DEBUG = false;

    private int discardPrefixLen = 5000;
    private int requiredSamples = 10000;
    private boolean randomFlipChoice = false;

    private Map/*<BayesNetNode, List<BayesNetNode>>*/ childMap;

    public GibbsSolver(BayesNet _bn) {
        super(_bn);
    }

    public GibbsSolver() {
        super();
    }

    /**
     * Set the number of samples that are discarded while the Markov chain
     * converges. Note that the total number of samples performed is still
     * that specified by setRequiredSamples, i.e. the total number that are
     * actually used is requiredSamples - discardPrefixLen.
     */
    public void setDiscardPrefixLen(int newLen) {
        discardPrefixLen = newLen;
    }

    /**
     * Set the number of samples performed
     */
    public void setRequiredSamples(int newSamples) {
        requiredSamples = newSamples;
    }

    /**
     * If set, the variable to be flipped in each sample will be chosen
     * uniformly at random rather than in systematic passes.
     */
    public void randomFlipChoice(boolean newVal) {
        randomFlipChoice = newVal;
    }


    /**
     * Generate an initial sample with all non-evidence variables selected
     * randomly.
     */
    private Assignment initialSample(BayesNetNodeSet bnset,
                                             Assignment evidence) {
        Assignment r = evidence;

        for (Iterator i =
                 bnset.getNodesWithTopologicalOrdering().iterator();
             i.hasNext();) {
            BayesNetNode node = (BayesNetNode) i.next();
            if (!evidence.contains(node.var)) {
                Comparable val = SampleUtils.sampleNode(node, r, true, null);
                r = new Assignment(r, node.var, val);
            }
        }

        return r;
    }

    /**
     * Resample the given node in the network, choosing a new node at random
     * with the probability distribution given by the current assignment of
     * the rest of the network.
     */
    private Assignment flipNode(Assignment last, BayesNetNode node) {
        Comparable val = SampleUtils.sampleNode(node, last, false, childMap);
        if (DEBUG) {
            System.out.println("Flipping node " + node.var + " to " + val);
        }

        return new Assignment(last, node.var, val);
    }

    public Function query (BayesNetNode node) {
        System.out.println("Query for " + node);
        System.out.println("Samples = " + requiredSamples);
        System.out.println("Discard = " + discardPrefixLen);


        FunctionVariable [] queryvar = new FunctionVariable[] {node.var};
        BayesNetNodeSet bnset = VESolver.GrabRelevantNodes(_bn.nodes,
                                                           queryvar,
                                                           _evidence);

        childMap = SampleUtils.buildChildMap(bnset);

        double [] prob = new double[node.var.domain.size()];

        Assignment x = initialSample(bnset, _evidence);

        int sampleCount = 0;


        loop:
        while (true) {
            for (int i = 0; i < bnset.size(); i++) {
                if (sampleCount >= requiredSamples) {
                    break loop;
                }

                BayesNetNode nodeToBeFlipped;
                // Node to be flipped is either sequential (given by the for
                // loop) or random
                if (randomFlipChoice) {
                    nodeToBeFlipped =
                        bnset.getNode(Random.random(bnset.size()));
                } else {
                    nodeToBeFlipped = bnset.getNode(i);
```

```java
                }

                // ...but it better not be an evidence variable.
                if (_evidence.contains(nodeToBeFlipped.var)) {
                    continue;
                }

                x = flipNode(x, nodeToBeFlipped);

                sampleCount += 1;
                if (sampleCount > discardPrefixLen) {
                    Comparable val = x.getAssignedValue(node.var);
                    int ind = node.var.domain.getIndex(val).i;
                    prob[ind] += 1;
                }
            }
        }

        return Compute.normalize(new Function(node.var, prob));
    }

    public String toString() {
        return "GibbsSolver";
    }

    public static void main(String[] args) {
        System.out.println("Prob(Burglary|JohnCalls=true, MaryCalls=true)");
        System.out.println("Burglary=TRUE AIMA: " + 0.284);

        final BayesNet bn = edu.mit.six825.bn.inputs.Nets.getBurglary();
        final Solver solver = new GibbsSolver();
        solver.setBayesNet(bn);
        //Solver solver = new EnumerationSolver(bn);
        // ...GibbsSamplerSolver(bn);
        // ...LikelihoodWeightingSolver(bn);
        // ...VariableEliminationSolver(bn);

        final FunctionVariable[] vars = new FunctionVariable[2];
        vars[0] = new FunctionVariable("JohnCalls");
        vars[1] = new FunctionVariable("MaryCalls");
        final Comparable[] vals = new Comparable[2];
        vals[0] = ComparableBoolean.TRUE;
        vals[1] = ComparableBoolean.TRUE;
        final Assignment evidence = new Assignment(new FunctionVariableSet(vars), vals
);
        solver.setEvidence(evidence);

        final BayesNetNode burgVar = bn.nodes.getNode("Burglary");
        final Function burgProb = solver.query(burgVar);
        System.out.println("Burglary=TRUE Calc.: " + burgProb);
    }

}
```

```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

import edu.mit.six825.bn.functiontable.*;
import edu.mit.six825.bn.bayesnet.*;
import edu.mit.six825.bn.inputs.*;
import techniques.utils.*;
import java.util.HashMap;

/**
 * An implementation of the likelihood weighting sampling algorithm
 * for Bayes nets.
 *
 * @author drkp
 */
public class LWSolver extends Solver {

    private static final double DEFAULT_REQUIRED_WEIGHT = 100.0;
    private double requiredWeight = DEFAULT_REQUIRED_WEIGHT;

    public LWSolver(BayesNet _bn) {
        super(_bn);
    }

    public LWSolver() {
        super();
    }

    public void setRequiredWeight(double newWeight) {
        requiredWeight = newWeight;
    }

    /**
     * A weighted sample -- just an assignment with a weight
     */
    public class WeightedParticle {
        public double weight;
        public Assignment assignment;

        public WeightedParticle(double _weight, Assignment _assignment) {
            assignment = _assignment;
            weight = _weight;
        }
    }

    /**
     * Generate a random sample satisfying the evidence, along with the
     * appropriate weight. This is implemented per procedure "LW-sample"
     * (Koller & Friedman p.330)
     */
    private WeightedParticle generateParticle(BayesNetNodeSet bnset) {
        double weight = 1.0;
        Assignment x = _evidence;       // sampled assignment so far
        List ordering = bnset.getNodesWithTopologicalOrdering();

//      System.out.println("Generating particle\n");
        for (Iterator i = ordering.iterator(); i.hasNext();) {
            BayesNetNode node = (BayesNetNode) i.next();

            if (_evidence.contains(node.var)) {
                double[] dist = SampleUtils.evaluateNodeGivenParents(node, x);
                Comparable val = _evidence.getAssignedValue(node.var);
                x = new Assignment(x, node.var, val);
                int ind = node.var.domain.getIndex(val).i;
                weight *= dist[ind];
//              System.out.println("Forcing " + node.var + " to " +
//                                 x.getAssignedValue(node.var));
//              System.out.println("Weight -> " + weight);
            } else {
                x = new Assignment(x, node.var,
                                   SampleUtils.sampleNode(node, x,
                                                          true, null));
//              System.out.println("Sampling " + node.var + " as " +
//                                 x.getAssignedValue(node.var));
            }
        }

        return new WeightedParticle(weight, x);
    }

    public Function query (BayesNetNode node) {
        System.out.println("Query for " + node);
        System.out.println("Required weight: " + requiredWeight);

        FunctionVariable [] queryvar = new FunctionVariable[] {node.var};
        BayesNetNodeSet bnset = VESolver.GrabRelevantNodes(_bn.nodes,
                                                            queryvar,
                                                            _evidence);

        double totalWeight = 0.0;
        int numSamples = 0;
        double [] prob = new double[node.var.domain.size()];
        while (numSamples < requiredWeight) {
            WeightedParticle particle = generateParticle(bnset);
            totalWeight += particle.weight;

            Comparable val = particle.assignment.getAssignedValue(node.var);
            int i = node.var.domain.getIndex(val).i;
            prob[i] += particle.weight;
            numSamples++;
        }

        // Normalize.
        for (int i = 0; i < node.var.domain.size(); i++) {
            prob[i] /= totalWeight;
        }

        return new Function(node.var, prob);
    }

    public String toString() {
        return "LWSolver";
    }

    public static void main(String[] args) {
        System.out.println("Prob(Burglary|JohnCalls=true, MaryCalls=true)");
        System.out.println("Burglary=TRUE AIMA: " + 0.284);

        final BayesNet bn = edu.mit.six825.bn.inputs.Nets.getBurglary();
        final Solver solver = new LWSolver();
        solver.setBayesNet(bn);
        //Solver solver = new EnumerationSolver(bn);
        // ...GibbsSamplerSolver(bn);
        // ...LikelihoodWeightingSolver(bn);
        // ...VariableEliminationSolver(bn);

        final FunctionVariable[] vars = new FunctionVariable[2];
        vars[0] = new FunctionVariable("JohnCalls");
        vars[1] = new FunctionVariable("MaryCalls");
```

```
        final Comparable[] vals = new Comparable[2];
        vals[0] = ComparableBoolean.TRUE;
        vals[1] = ComparableBoolean.TRUE;
        final Assignment evidence = new Assignment(new FunctionVariableSet(vars), vals
);
        solver.setEvidence(evidence);

        final BayesNetNode burgVar = bn.nodes.getNode("Burglary");
        final Function burgProb = solver.query(burgVar);
        System.out.println("Burglary=TRUE Calc.: " + burgProb);
    }

}
```

```java
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.Iterator;
import java.util.ArrayList;

import edu.mit.six825.bn.functiontable.*;
import edu.mit.six825.bn.bayesnet.*;
import edu.mit.six825.bn.inputs.*;
import techniques.utils.*;



/**
 * Some utilities for implementing sampling-based solvers
 *
 * @author drkp
 */
public class SampleUtils {
    private static final boolean DEBUG = false;

    private SampleUtils() {

    }

    public static Map buildChildMap(BayesNetNodeSet bnset) {
        Map childMap = new HashMap();
        for (Iterator it = bnset.iterator(); it.hasNext();) {
            BayesNetNode node = (BayesNetNode) it.next();
            childMap.put(node, new ArrayList());
        }

        for (Iterator it = bnset.iterator(); it.hasNext();) {
            BayesNetNode parent = (BayesNetNode) it.next();
            for (Iterator it2 = bnset.iterator(); it2.hasNext();) {
                BayesNetNode child = (BayesNetNode) it2.next();
                if (child.parents.contains(parent)) {
                    List lst = (List) childMap.get(parent);
                    lst.add(child);
                }

            }
        }

        return childMap;
    }


    /**
     * Return the probability of each value in the node's variable's domain,
     * given an assignment that includes its parents values.
     */
    public static double [] evaluateNodeGivenParents(BayesNetNode node,
                                                     Assignment evidence) {
        double [] entries = new double[node.var.domain.size()];

        for (int i = 0; i < node.var.domain.size(); i++) {
            Comparable val = node.var.domain.getValue(i);
            Assignment fullAss = new Assignment(evidence, node.var, val);
            entries[i] = node.cpt.evaluate(fullAss);
        }

        return entries;
    }

    /**
     * Return the probability of each value in the node's variable's domain,
     * consistent with the current assignments of all of its children. This
     * requires that the assignment provided includes the full Markov blanket
     * (parents, children, and children's parents) of the given node.
     *
     * Implemented per formula (14.11) of AIMA.
     */
    public static double [] evaluateNodeGivenMarkovBlanket(BayesNetNode node,
                                                           Assignment as,
                                                           Map childMap) {

        Function cpt = node.cpt;
        double sum = 0;

        if (DEBUG) {
            System.out.println("Evaluating node " + node.var);
            System.out.println("Assignment is " + as);
        }

        double [] entries = new double[node.var.domain.size()];

        for (int i = 0; i < node.var.domain.size(); i++) {
            Comparable val = node.var.domain.getValue(i);
            Assignment fullAss = new Assignment(as, node.var, val);
            if (DEBUG) {
                System.out.println(" Considering assignment " + val);
            }

            entries[i] = cpt.evaluate(fullAss);
            if (DEBUG) {
                System.out.println("  CPT evaluates to " + entries[i]);
            }

            // Multiply by the probabilities of each child
            for (Iterator it2 = ((List)childMap.get(node)).iterator();
                 it2.hasNext();) {
                BayesNetNode child = (BayesNetNode) it2.next();

                double childProb = child.cpt.evaluate(fullAss);
                entries[i] *= childProb;
                if (DEBUG) {
                    System.out.println("  Child " + child.var +
                                       " gives prob " +
                                       childProb);
                }
            }
            sum += entries[i];
        }

        for (int i = 0; i < node.var.domain.size(); i++) {
            entries[i] /= sum;
            if (DEBUG) {
                System.out.println("Final: " + entries[i]);
            }
        }

        return entries;
    }

    /**
     * Select a value of a node's variable according to the probability
     * distribution specified by the given assignment.
     *
     * If useOnlyParents is specified, then the sampling is performed based
```

```
     * solely on the distribution given by the parents' values; otherwise it
     * is performed based on the full Markov blanket. So setting this false is
     * useful when generating an initial sample, and true useful when sampling
     * given a full assignment.
     */
    public static Comparable sampleNode(BayesNetNode node,
                                        Assignment evidence,
                                        boolean useOnlyParents,
                                        Map childMap) {
        double[] dist;
        if (useOnlyParents) {
            dist = evaluateNodeGivenParents(node, evidence);
        } else {
            dist = evaluateNodeGivenMarkovBlanket(node, evidence, childMap);
        }

        int r = Random.sampleCompleteProbDist(dist);
        return node.var.domain.getValue(r);
    }
}
```

```java
import edu.mit.six825.bn.functiontable.*;
import edu.mit.six825.bn.bayesnet.*;
import edu.mit.six825.bn.inputs.*;

import java.util.*;

/**
 * A wrapper that lets us run any given solver in any given
 * configuration against any of the specified queries. See below for
 * invocation syntax.
 *
 * @author nocturne
 */
public class RunTest {

    public static Solver PickSolver (String ss) {
        if (ss.equals("enum")) {
            return(new EnumerationSolver());
        } else if (ss.equals("ve")) {
            return(new VESolver());
        } else if (ss.equals("lw")) {
            return(new LWSolver());
        } else if (ss.equals("gibbs")) {
            return(new GibbsSolver());
        } else {
            throw new IllegalArgumentException("Unknown solver type " + ss);
        }
    }

    public static VESolver.ElimOrderer PickElim (String ss) {
        if (ss.equals("dumb")) {
            System.out.println("Using dumb orderer.");
            return((new VESolver.DumbOrder()));
        } else if (ss.equals("random")) {
            System.out.println("Using random orderer.");
            return((new VESolver.RandomOrder()));
        } else if (ss.equals("greedy")) {
            System.out.println("Using greedy orderer.");
            return((new VESolver.GreedyOrder()));
        } else {
            throw new IllegalArgumentException("Unknown elim type " + ss);
        }
    }

    /**
     * Intended argument syntax:
     *    --solver ( enum | ve | lw )
     *    --elim   ( dumb | random | greedy )
     *    --weight (num)
     *    --samples (num)
     *    --discard (prefixlen)
     *    --randorder
     *    ( 1.[123] | 2.[1234] )
     *
     * specifying an elimination orderer is only relevant if you're
     * using the VE solver.
     * specifying a required weight is only relevant for LW solving.
     * number of samples, discard prefix len, and random ordering are
     * only relevant to Gibbs sampling.
     *
     * Sample args: "--solver ve --elim dumb 2.2"
     *              "--solver lw 1.3 --weight 100"
     *              "--solver gibbs samples 1000 --discard 500 --randorder"
     *
     */
    public static void main(String[] args) {

        Solver      solver  = new EnumerationSolver();
        VESolver.ElimOrderer elimord = new VESolver.DumbOrder();
        String   whichquery = "1.1";
        double   requiredWeight = 100.0;
        int      requiredSamples = 1000;
        int      discardPrefixLen = 500;
        boolean  randOrder = false;

        for (int i=0; i < args.length; i++) {
            String arg = args[i];
            if (arg.equals("--solver")) {
                i++;
                if (i < args.length) {
                    solver = PickSolver(args[i]);
                } else {
                    throw new IllegalArgumentException("--solver requires"
                                                       + " an argument");
                }
            } else if (arg.equals("--elim")) {
                i++;
                if (i < args.length) {
                    elimord = PickElim(args[i]);
                } else {
                    throw new IllegalArgumentException("--elim requires"
                                                       + " an argument");
                }
            } else if (arg.equals("--weight")) {
                i++;
                if (i < args.length) {
                    requiredWeight = Double.parseDouble(args[i]);
                } else {
                    throw new IllegalArgumentException("--weight requires"
                                                       + " an argument");
                }
            } else if (arg.equals("--samples")) {
                i++;
                if (i < args.length) {
                    requiredSamples = Integer.parseInt(args[i]);
                } else {
                    throw new IllegalArgumentException("--weight requires"
                                                       + " an argument");
                }
            } else if (arg.equals("--discard")) {
                i++;
                if (i < args.length) {
                    discardPrefixLen = Integer.parseInt(args[i]);
                } else {
                    throw new IllegalArgumentException("--discard requires"
                                                       + " an argument");
                }
            } else if (arg.equals("--randorder")) {
                randOrder = true;
            } else {
                whichquery = arg;
            }
        }

        if (solver.getClass() == VESolver.class) {
            ((VESolver)solver).setOrderer(elimord);
        } else if (solver.getClass() == LWSolver.class) {
            ((LWSolver)solver).setRequiredWeight(requiredWeight);
        } else if (solver.getClass() == GibbsSolver.class) {
            ((GibbsSolver)solver).setRequiredSamples(requiredSamples);
```

```java
            ((GibbsSolver)solver).setDiscardPrefixLen(discardPrefixLen);
            ((GibbsSolver)solver).randomFlipChoice(randOrder);
        }

        BayesNet   bnet;
        String []  evidencearr;
        String     query;

        if (whichquery.equals("1.1")) {
            bnet = Nets.getBurglary();
            evidencearr = new String[] {"JohnCalls", "true",
                                        "MaryCalls", "true"};
            query = "Burglary";
        } else if (whichquery.equals("1.2")) {
            bnet = Nets.getBurglary();
            evidencearr = new String[] {"Burglary",  "true",
                                        "JohnCalls", "true"};
            query = "Earthquake";
        } else if (whichquery.equals("1.3")) {
            bnet = Nets.getInsurance();
            evidencearr = new String[] {"Age", "Adolescent",
                                        "Airbag", "False",
                                        "Mileage", "TwentyThou"};
            query = "PropCost";
        } else if (whichquery.equals("2.1")) {
            bnet = Nets.getInsurance();
            evidencearr = new String[] {"Age", "Adolescent",
                                        "Airbag", "False",
                                        "MakeModel", "Luxury",
                                        "Mileage", "TwentyThou"};
            query = "PropCost";
        } else if (whichquery.equals("2.2")) {
            bnet = Nets.getInsurance();
            evidencearr = new String[] {"Age", "Adolescent",
                                        "Airbag", "False",
                                        "GoodStudent", "True",
                                        "Mileage", "TwentyThou"};
            query = "PropCost";
        } else if (whichquery.equals("2.3")) {
            bnet = Nets.getCarpo();
            evidencearr = new String[] {"N116", "0",
                                        "N41", "2",
                                        "N84", "1"};
            query = "N104";
        } else if (whichquery.equals("2.4")) {
            bnet = Nets.getCarpo();
            evidencearr = new String[] {"N116", "0",
                                        "N152", "1",
                                        "N43", "1"};
            query = "N73";
        } else {
            throw new IllegalArgumentException("Unknown query "
                                               + whichquery);
        }

        solver.setBayesNet(bnet);

        Assignment evidence;
        try {
            evidence =
                VESolver.ConstructEvidence(bnet.nodes, evidencearr);

            solver.setEvidence(evidence);
        }
        catch (IllegalArgumentException e) {
            System.out.println("ERROR: " + e);
            return;
        }

        BayesNetNode querynode;
        try {
            querynode = bnet.nodes.getNode(query);
        }
        catch (IllegalArgumentException e) {
            System.out.println("ERROR: " + e);
            return;
        }

        long starttime = System.currentTimeMillis();
        Function answer = solver.query(querynode);
        long endtime = System.currentTimeMillis();

        System.out.println("Query:    " + query);
        System.out.println("Evidence: " + evidence);
        System.out.println(answer);
        System.out.println("Solver time elapsed: " + (endtime-starttime) +
                           " milliseconds.");
    }
}
```