OpenRules Decision Model for
DMCommunity June-2024 Challenge “Smart Marriages”

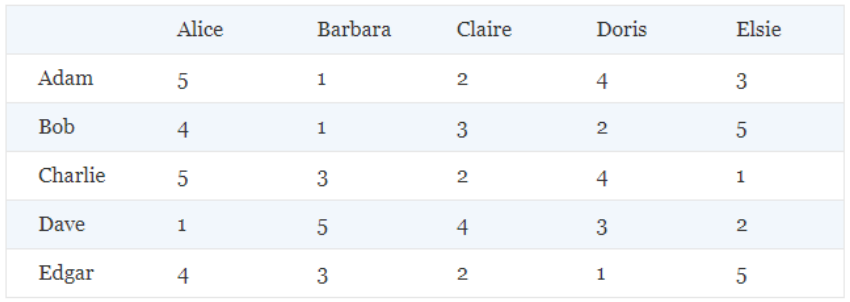
Problem

This challenge deals with the famous stable marriage problem formulated as follows:

“Given n men and n women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable.”

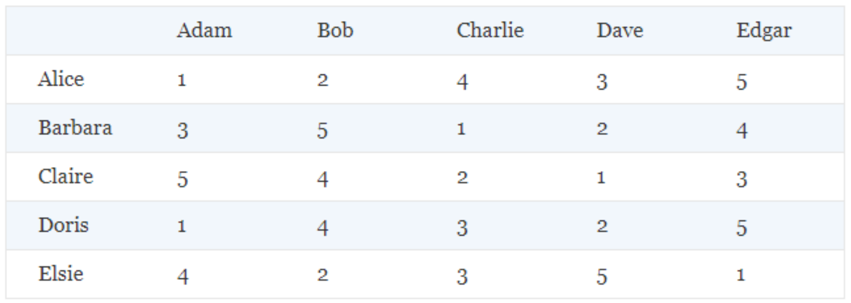
Here is an example of the problem:

How Men Rank Women:



	Alice	Barbara	Claire	Doris	Elsie
Adam	5	1	2	4	3
Bob	4	1	3	2	5
Charlie	5	3	2	4	1
Dave	1	5	4	3	2
Edgar	4	3	2	1	5

How Women Rank Men:



	Adam	Bob	Charlie	Dave	Edgar
Alice	1	2	4	3	5
Barbara	3	5	1	2	4
Claire	5	4	2	1	3
Doris	1	4	3	2	5
Elsie	4	2	3	5	1

We need to build a decision model capable of pairing up a group of men and women so that the resulting marriages are stable. A good analysis of the problem is provided in the recent presentation given by Dr. Guido Tack.

Solution

Business Part

First, I wanted to create a business environment for this problem using OpenRules Decision Manager.

I started with the creation of a business glossary:

Glossary glossary			
Variables	**Business Concept**	**Attribute**	**Type**
Men	**Marriages**	men	Person[]
Women		women	Person[]
Person Name	**Person**	name	String
Person Preferences		preferences	int[]

Then I created test data to represent the above instance of the problem with 5 men and 5 women:

DecisionData Person men5	
Person Name	**Person Preferences**
Adam	5,1,2,4,3
Bob	4,1,3,2,5
Charlie	5,3,2,4,1
Dave	1,5,4,3,2
Edgar	4,3,2,1,5

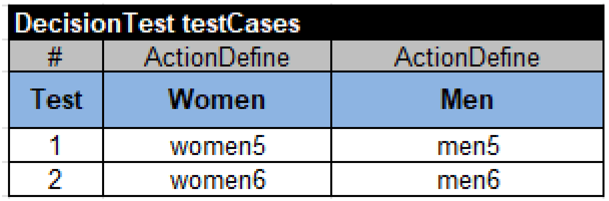
DecisionData Person women5	
Person Name	**Person Preferences**
Alice	1,2,4,3,5
Barbara	3,5,1,2,4
Claire	5,4,2,1,3
Doris	1,4,3,2,5
Elsie	4,2,3,5,1

I also decided to add an example with 6 men and 6 women from Guido's presentation (he mentioned that was not solved):

DecisionData Person men6	
Person Name	**Person Preferences**
Adam	1,2,4,5,6,3
Bob	4,2,5,3,1,6
Charlie	2,5,4,6,3,1
Dave	3,4,2,6,5,1
Edgar	6,3,2,1,5,4
Fred	4,6,3,2,1,5

DecisionData Person women6	
Person Name	**Person Preferences**
Alice	5,2,6,3,1,4
Barbara	6,5,1,2,4,3
Claire	3,2,1,4,5,6
Doris	2,4,1,3,5,6
Elsie	3,5,4,2,1,6
Fiona	5,1,6,4,2,3

I added these samples as test cases for my OpenRules-based decision model:



DecisionTest testCases		
#	ActionDefine	ActionDefine
Test	**Women**	**Men**
1	women5	men5
2	women6	men6

It allowed me to test my environment and see it visually using OpenRules Explorer/Debugger before I started thinking about how to represent and solve the problem constraints. I already could even deploy my decision model on the cloud and any desirable way.

Technical Part

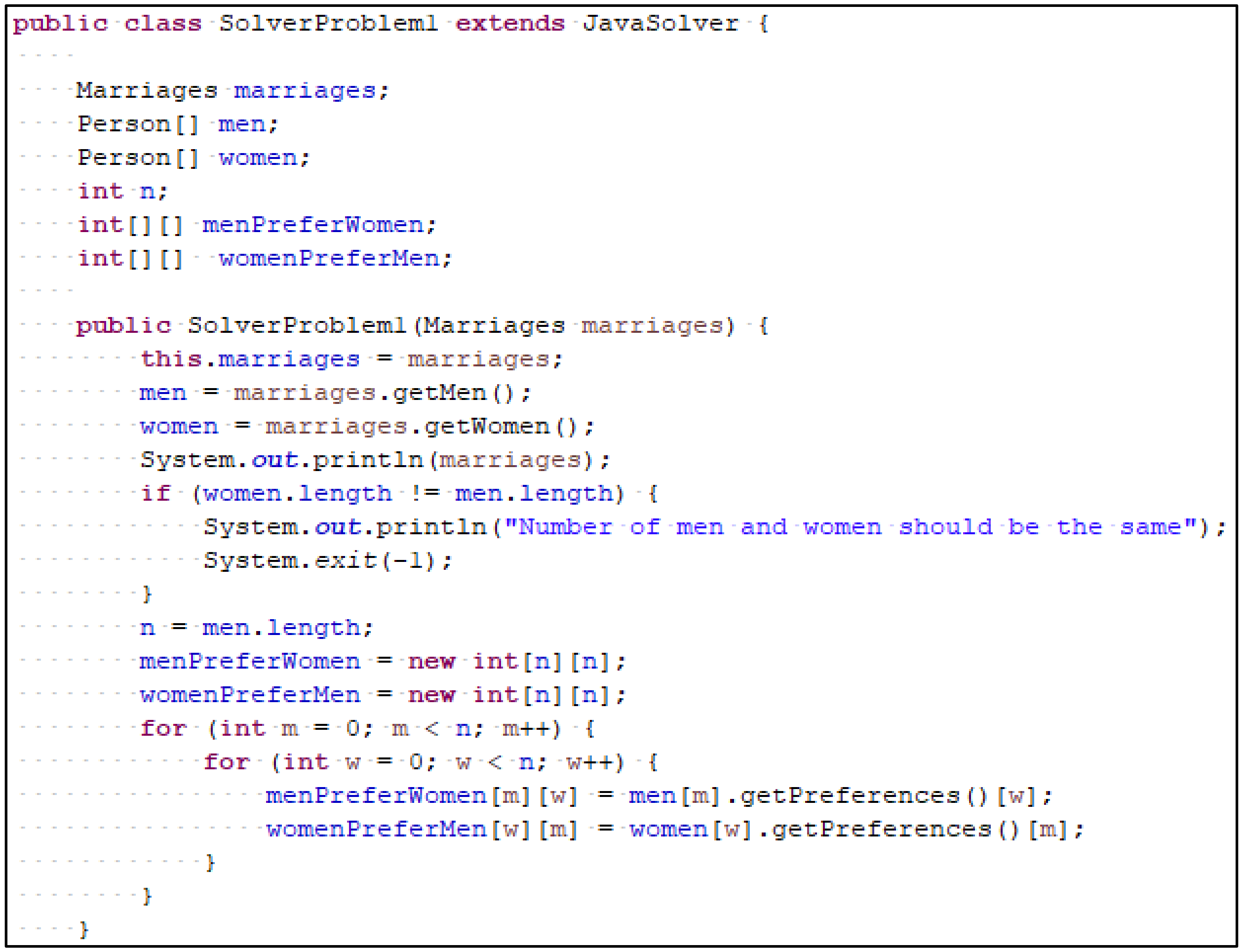
Below I will describe two different approaches I used to solve this problem.

Approach 1

Dr. Guido Tack has already provided a very good analysis of the problem describing the famous Gale-Shapley algorithm and providing different implementions using MiniZinc. Guido's attempts to find a "fair" solution were very enlightening but as he admitted they were not very successful. These considerations pushed me to consider a more simple approach.

Instead of implementing complex marriage stability constraints (that did not help to produce a good fair solution anyway) we could just represent simple constraints that each person should marry one and only one person of the opposite gender. Then we may summarize preferences for all pairs "man-woman" and "woman-man" and consider this sum as our optimization objective. When we minimize this objective, we will receive a solution with the most top preferences being satisfied. Here is how I implemented this approach using OpenRules "Rule Solver".

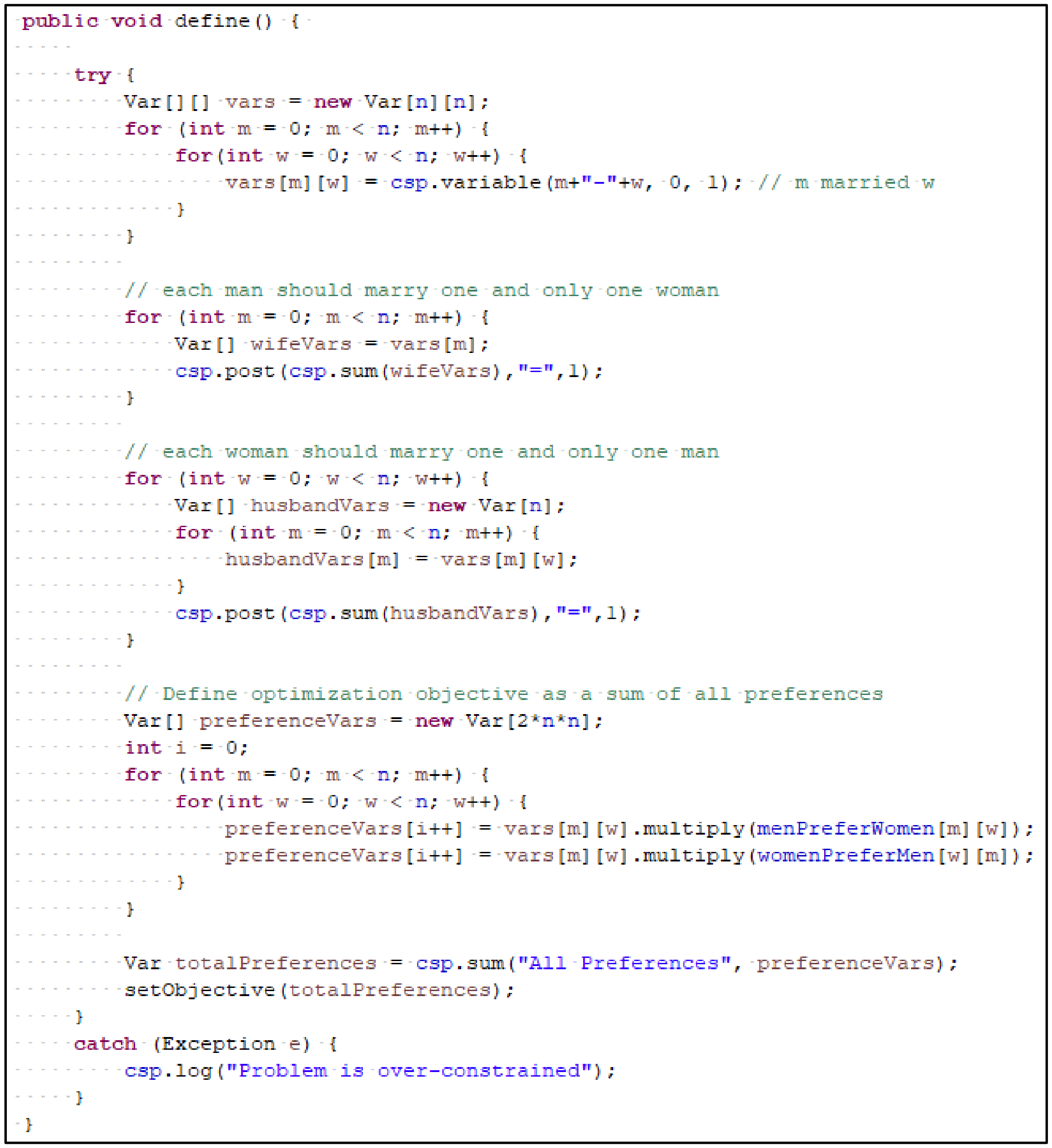
I implemented this approach inside a Java class "SolverProblem1" inherited from the standard JavaSolver class JavaSolver. Here is the constructor for this class:



```java
public class SolverProblem1 extends JavaSolver {

    Marriages marriages;
    Person[] men;
    Person[] women;
    int n;
    int[][] menPreferWomen;
    int[][]  womenPreferMen;

    public SolverProblem1(Marriages marriages) {
        this.marriages = marriages;
        men = marriages.getMen();
        women = marriages.getWomen();
        System.out.println(marriages);
        if (women.length != men.length) {
            System.out.println("Number of men and women should be the same");
            System.exit(-1);
        }
        n = men.length;
        menPreferWomen = new int[n][n];
        womenPreferMen = new int[n][n];
        for (int m = 0; m < n; m++) {
            for (int w = 0; w < n; w++) {
                menPreferWomen[m][w] = men[m].getPreferences()[w];
                womenPreferMen[w][m] = women[w].getPreferences()[m];
            }
        }
    }
```

It simply takes the arrays of *men* and *women* from the business concept “Marriages” and defines two 2-dimensional arrays *menPreferWomen* and *womenPreferMen*.

Then I concentrated on adding method *define()* to define all unknown decision variables, constraints, and the optimization objective. Here it goes:



```java
public void define() {

    try {
        Var[][] vars = new Var[n][n];
        for (int m = 0; m < n; m++) {
            for(int w = 0; w < n; w++) {
                vars[m][w] = csp.variable(m+"-"+w, 0, 1); // m married w
            }
        }

        // each man should marry one and only one woman
        for (int m = 0; m < n; m++) {
            Var[] wifeVars = vars[m];
            csp.post(csp.sum(wifeVars),"=",1);
        }

        // each woman should marry one and only one man
        for (int w = 0; w < n; w++) {
            Var[] husbandVars = new Var[n];
            for (int m = 0; m < n; m++) {
                husbandVars[m] = vars[m][w];
            }
            csp.post(csp.sum(husbandVars),"=",1);
        }

        // Define optimization objective as a sum of all preferences
        Var[] preferenceVars = new Var[2*n*n];
        int i = 0;
        for (int m = 0; m < n; m++) {
            for(int w = 0; w < n; w++) {
                preferenceVars[i++] = vars[m][w].multiply(menPreferWomen[m][w]);
                preferenceVars[i++] = vars[m][w].multiply(womenPreferMen[w][m]);
            }
        }

        Var totalPreferences = csp.sum("All Preferences", preferenceVars);
        setObjective(totalPreferences);
    }
    catch (Exception e) {
        csp.log("Problem is over-constrained");
    }
}
```

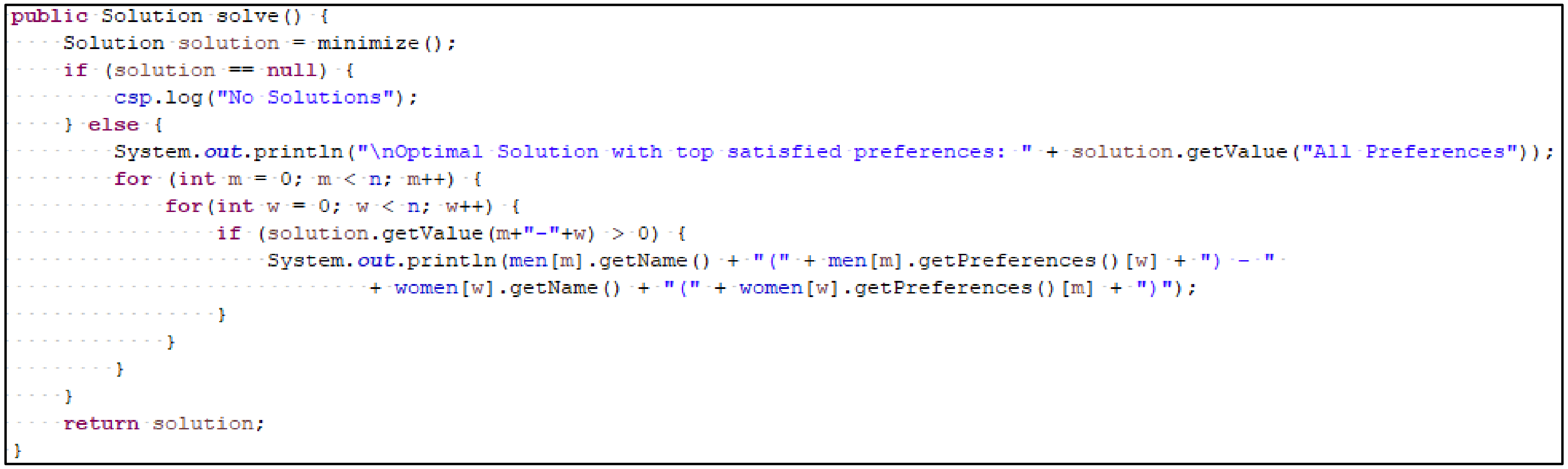
First, I created a 2-dimensional array of Boolean constrained variable *vars* for each pair "man-woman".

To state that *each man should marry one and only one woman* I selected a sub-array "wifeVars" for each man and posted the constraints "sum of wifeVars = 1".

To state that *each woman should marry one and only one man* I defined a sub-array "husbandVars" for each woman and posted the constraints "sum of husbandVars = 1".

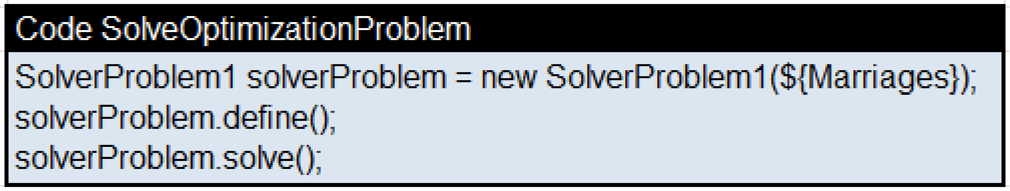
Finally, I defined the optimization objective as a sum of All Preferences placed in the array of constrained variable *preferenceVars*.

To solve the problem I could rely on the standard JavaSolver method "*minimize()*", but I wanted to print the solution nicely, so I added the following method "solve()":



```java
public Solution solve() {
    Solution solution = minimize();
    if (solution == null) {
        csp.log("No Solutions");
    } else {
        System.out.println("\nOptimal Solution with top satisfied preferences: " + solution.getValue("All Preferences"));
        for (int m = 0; m < n; m++) {
            for(int w = 0; w < n; w++) {
                if (solution.getValue(m+"-"+w) > 0) {
                    System.out.println(men[m].getName() + "(" + men[m].getPreferences()[w] + ") - "
                            + women[w].getName() + "(" + women[w].getPreferences()[m] + ")");
                }
            }
        }
    }
    return solution;
}
```

To invoke this Java method from my decision model, I used the following Excel table:

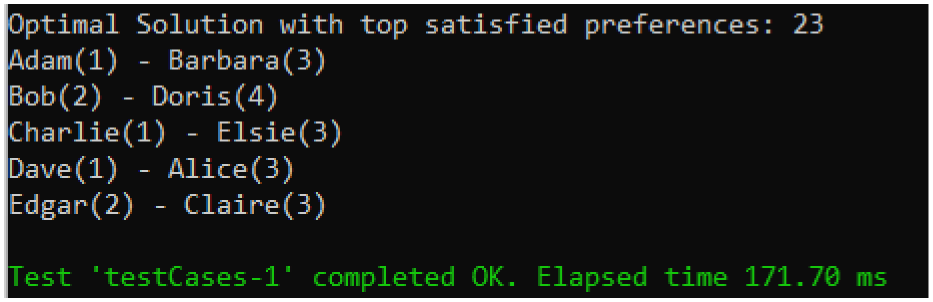


Code SolveOptimizationProblem

SolverProblem1 solverProblem = new SolverProblem1(${Marriages});
solverProblem.define();
solverProblem.solve();

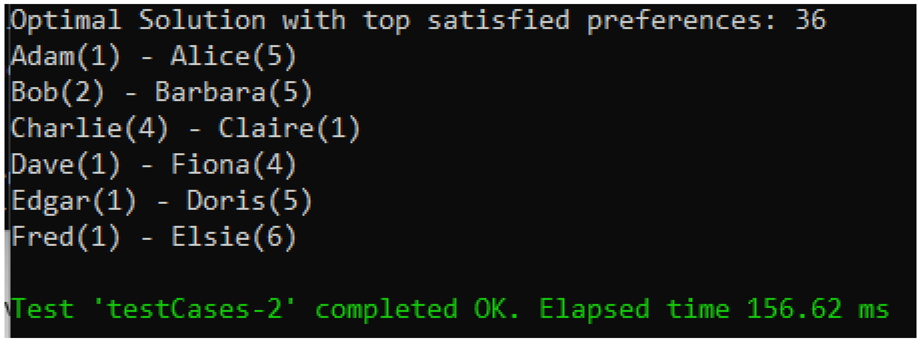
When I executed my decision model I received the following solution:

Test Case 1



```
Optimal Solution with top satisfied preferences: 23
Adam(1) - Barbara(3)
Bob(2) - Doris(4)
Charlie(1) - Elsie(3)
Dave(1) - Alice(3)
Edgar(2) - Claire(3)

Test 'testCases-1' completed OK. Elapsed time 171.70 ms
```

Test Case 2



```
Optimal Solution with top satisfied preferences: 36
Adam(1) - Alice(5)
Bob(2) - Barbara(5)
Charlie(4) - Claire(1)
Dave(1) - Fiona(4)
Edgar(1) - Doris(5)
Fred(1) - Elsie(6)

Test 'testCases-2' completed OK. Elapsed time 156.62 ms
```

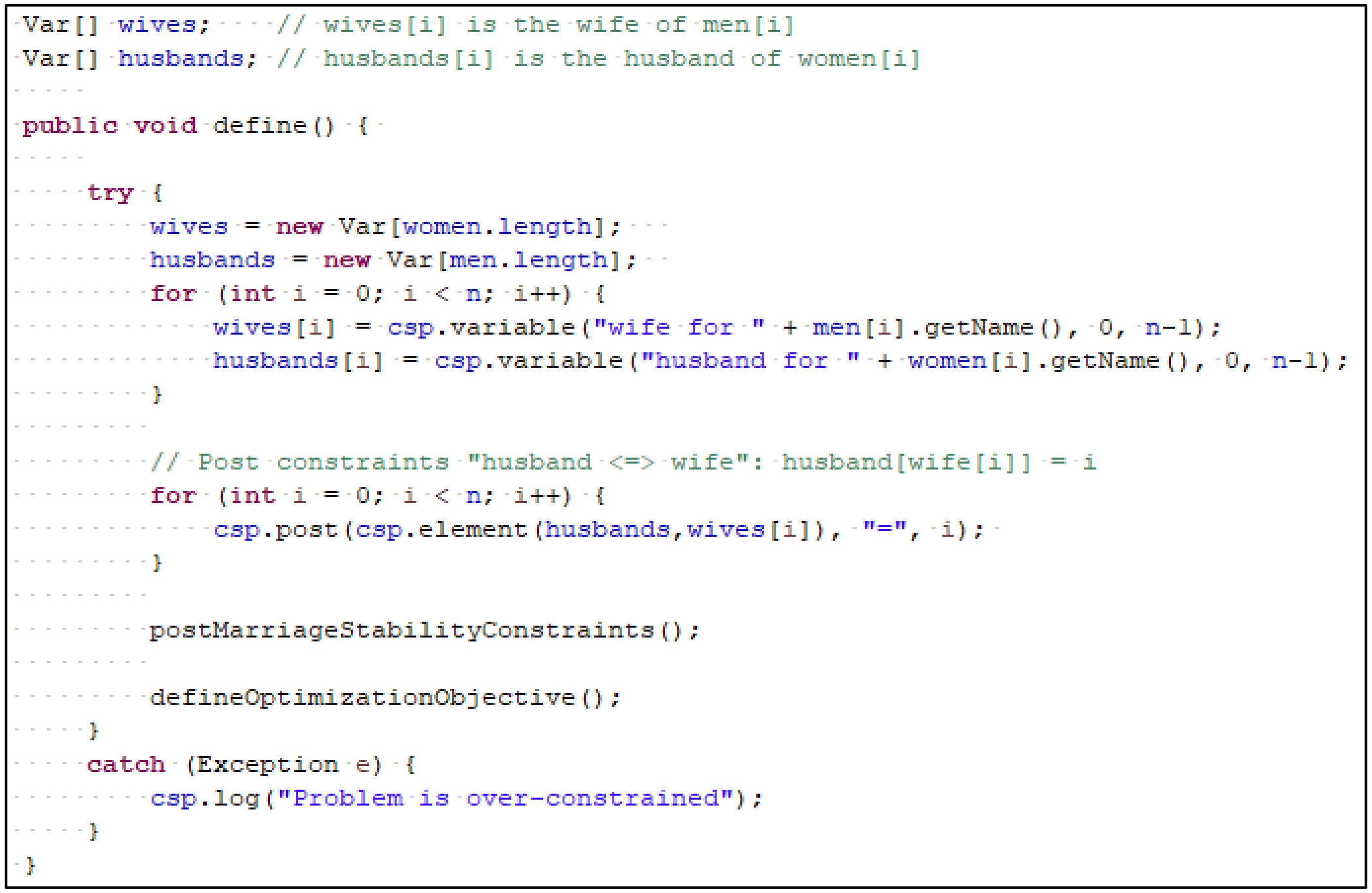
You can read it as “Adam selects Alice with preference 1 and Alice selects Adam with preference 5”, etc.

As we can see, Approach 1 gave us a solution with the maximum possible preferences being satisfied.

Approach 2

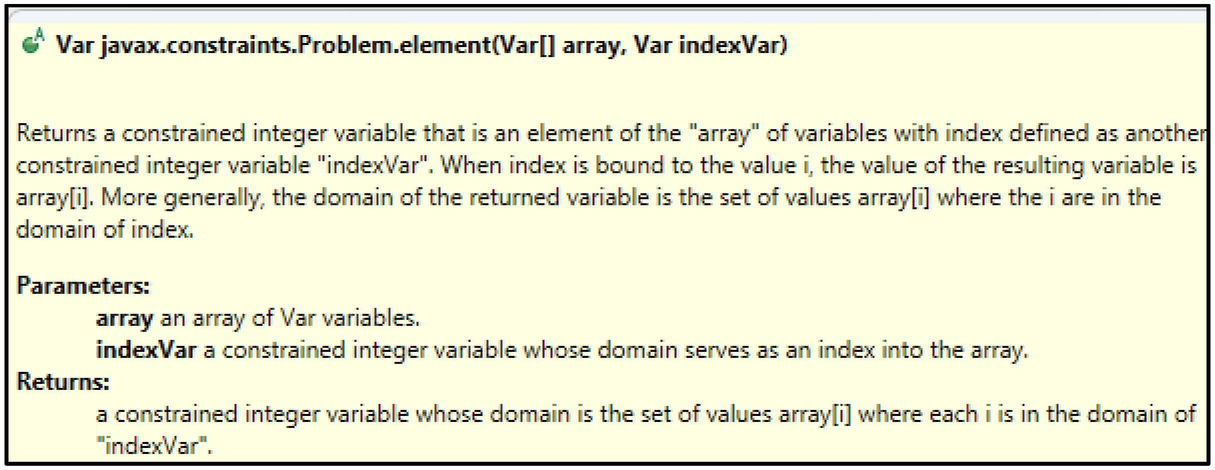
I still wanted to implement an approach similar to Dr. Guido Tack's solutions with explicitly specified marriage stability constraints and compare the results with my Approach 1. To do this, I decided to simply create another Java class “SolverProblem2” and invoke using a similar Excel table “SolveOptimizationProblem” but with SolverProblem2 instead of SolverProblem1.

This time I decided to introduce two arrays of constrained integer variable ***wives*** and ***husbands*** of the type Var[] with yet unknown variables that correspond to indexes in the array *women* and *men*. For instance, *husbands[2]* is yet unknown variable that represents an index of a man in the array *men* (a husband of *women[2])*.



```java
Var[] wives;    // wives[i] is the wife of men[i]
Var[] husbands; // husbands[i] is the husband of women[i]

public void define() {

    try {
        wives = new Var[women.length];
        husbands = new Var[men.length];
        for (int i = 0; i < n; i++) {
            wives[i] = csp.variable("wife for " + men[i].getName(), 0, n-1);
            husbands[i] = csp.variable("husband for " + women[i].getName(), 0, n-1);
        }

        // Post constraints "husband <=> wife": husband[wife[i]] = i
        for (int i = 0; i < n; i++) {
            csp.post(csp.element(husbands,wives[i]), "=", i);
        }

        postMarriageStabilityConstraints();

        defineOptimizationObjective();
    }
    catch (Exception e) {
        csp.log("Problem is over-constrained");
    }
}
```

To express constraints "*if a man is a husband of a woman then the woman is a wife of the man*", I used the standard Java Solver's constraint "element" specified as follows:



Var javax.constraints.Problem.element(Var[] array, Var indexVar)

Returns a constrained integer variable that is an element of the "array" of variables with index defined as another constrained integer variable "indexVar". When index is bound to the value i, the value of the resulting variable is array[i]. More generally, the domain of the returned variable is the set of values array[i] where the i are in the domain of index.

Parameters:
array an array of Var variables.
indexVar a constrained integer variable whose domain serves as an index into the array.
Returns:
a constrained integer variable whose domain is the set of values array[i] where each i is in the domain of "indexVar".

In our case, it creates a constraint variable *csp.element(husbands,wives[i])* that refers to the element in the arrays *husbands* with the index *wives[i]*. Our constraints state that this variable should be equal to the current index *i* (which is used to select a wife as *wives[i]*).

Note. I admit that this constraint is not easy to understand. If instead of Java API (without overloaded operators) I used a specialized constraint language such as MiniZinc, this constraint would look much nicer like in Guido's presentation:

```
constraint forall (m in Men) (husband[wife[m]]=m);
```

To express even more complex marriage stability constraints, I created a special method *postMarriageStabilityConstraints()* and added it to the above method *define()*. These constraints should express the following logic:

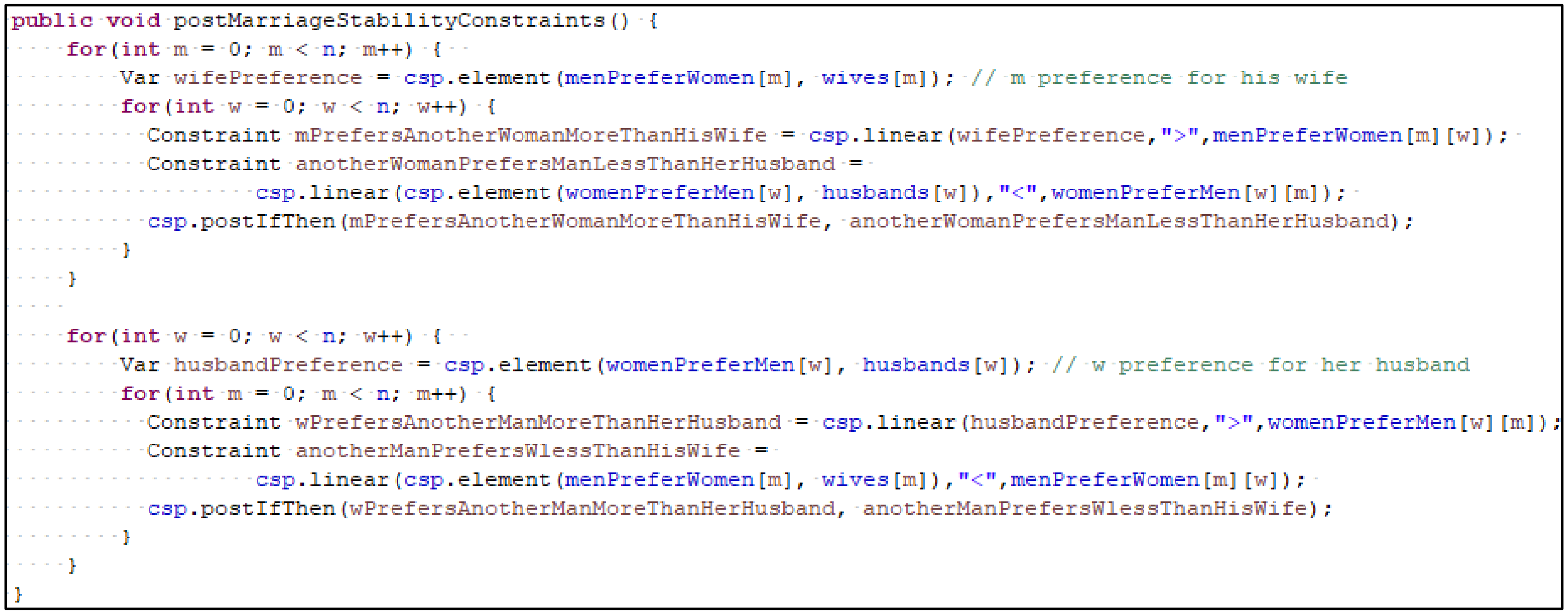
For all men:

"If a man prefers another woman more than his wife, then that other woman should prefer this man less than her husband."

For all women:

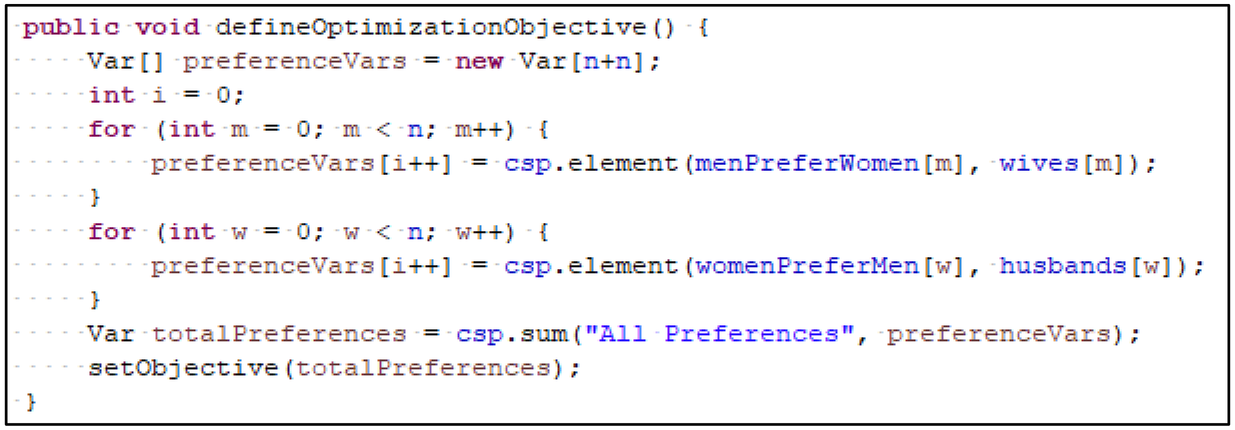
"If a woman prefers another man more than her husband, then that other man should prefer this woman less than his wife."

I again used the standard constraint "element" to express these constraints:



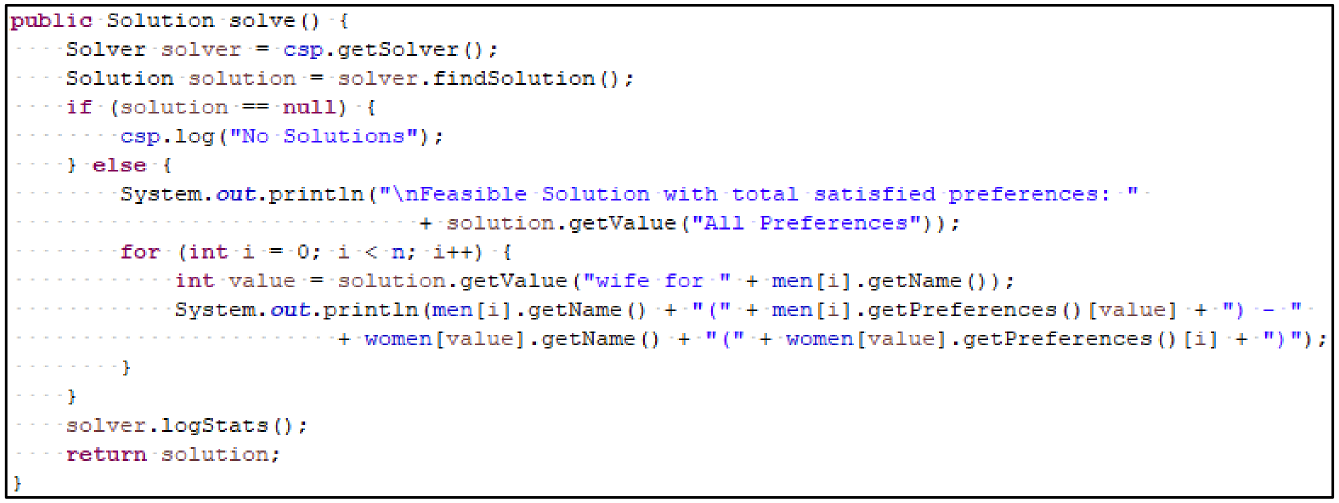
```java
public void postMarriageStabilityConstraints() {
    for(int m = 0; m < n; m++) {
        Var wifePreference = csp.element(menPreferWomen[m], wives[m]); // m preference for his wife
        for(int w = 0; w < n; w++) {
          Constraint mPrefersAnotherWomanMoreThanHisWife = csp.linear(wifePreference,">",menPreferWomen[m][w]);
          Constraint anotherWomanPrefersManLessThanHerHusband =
                  csp.linear(csp.element(womenPreferMen[w], husbands[w]),"<",womenPreferMen[w][m]);
          csp.postIfThen(mPrefersAnotherWomanMoreThanHisWife, anotherWomanPrefersManLessThanHerHusband);
        }
    }

    for(int w = 0; w < n; w++) {
        Var husbandPreference = csp.element(womenPreferMen[w], husbands[w]); // w preference for her husband
        for(int m = 0; m < n; m++) {
          Constraint wPrefersAnotherManMoreThanHerHusband = csp.linear(husbandPreference,">",womenPreferMen[w][m]);
          Constraint anotherManPrefersWlessThanHisWife =
                  csp.linear(csp.element(menPreferWomen[m], wives[m]),"<",menPreferWomen[m][w]);
          csp.postIfThen(wPrefersAnotherManMoreThanHerHusband, anotherManPrefersWlessThanHisWife);
        }
    }
}
```

To define the optimization objective I used the following method:



```java
public void defineOptimizationObjective() {
    Var[] preferenceVars = new Var[n+n];
    int i = 0;
    for (int m = 0; m < n; m++) {
        preferenceVars[i++] = csp.element(menPreferWomen[m], wives[m]);
    }
    for (int w = 0; w < n; w++) {
        preferenceVars[i++] = csp.element(womenPreferMen[w], husbands[w]);
    }
    Var totalPreferences = csp.sum("All Preferences", preferenceVars);
    setObjective(totalPreferences);
}
```

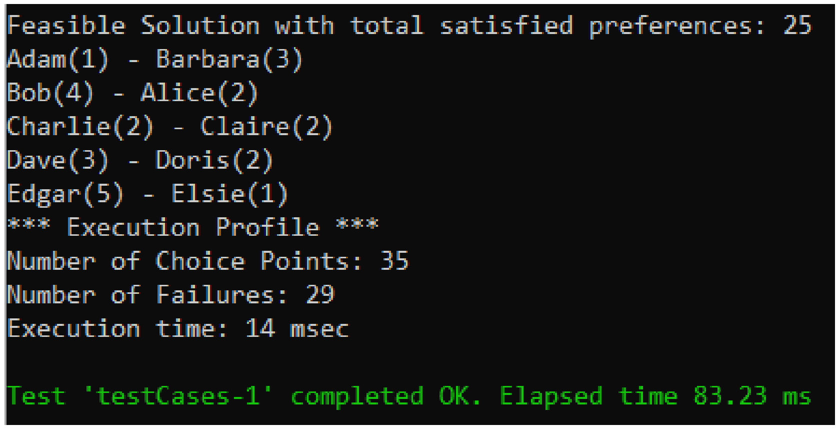
To find a feasible solution of this problem, I could rely on the default JavaSolver method *solve()*. However, to print a solution nicely I overload it as follows:



```java
public Solution solve() {
    Solver solver = csp.getSolver();
    Solution solution = solver.findSolution();
    if (solution == null) {
        csp.log("No Solutions");
    } else {
        System.out.println("\nFeasible Solution with total satisfied preferences: "
                              + solution.getValue("All Preferences"));
        for (int i = 0; i < n; i++) {
            int value = solution.getValue("wife for " + men[i].getName());
            System.out.println(men[i].getName() + "(" + men[i].getPreferences()[value] + ") - "
                          + women[value].getName() + "(" + women[value].getPreferences()[i] + ")");
        }
    }
    solver.logStats();
    return solution;
}
```

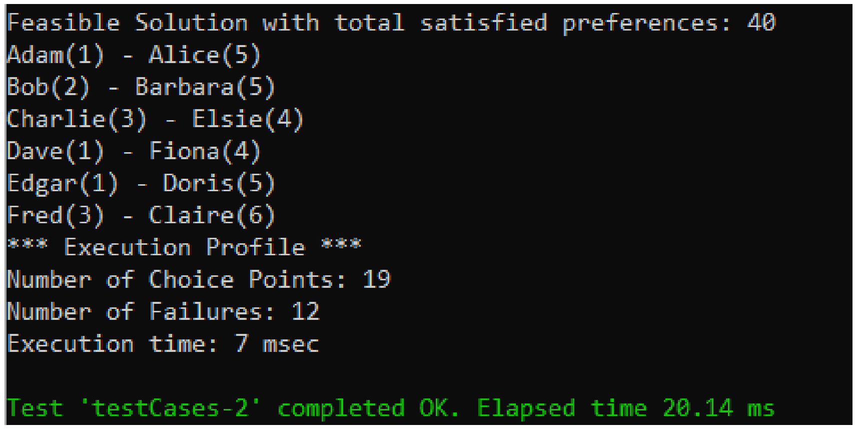
When I executed this decision model with SolverProblem2, I received the following feasible solutions for test cases 1 and 2:

Test Case 1



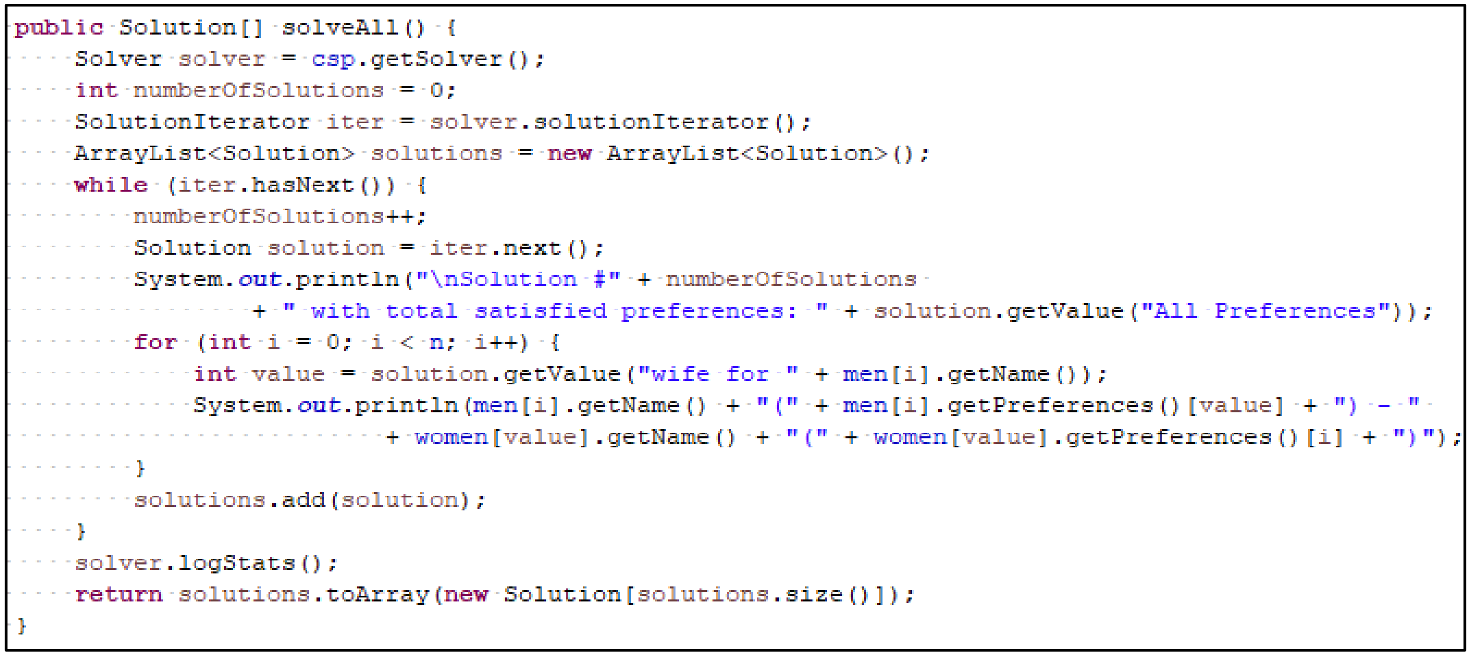
```
Feasible Solution with total satisfied preferences: 25
Adam(1) - Barbara(3)
Bob(4) - Alice(2)
Charlie(2) - Claire(2)
Dave(3) - Doris(2)
Edgar(5) - Elsie(1)
*** Execution Profile ***
Number of Choice Points: 35
Number of Failures: 29
Execution time: 14 msec

Test 'testCases-1' completed OK. Elapsed time 83.23 ms
```

Test Case 2



```
Feasible Solution with total satisfied preferences: 40
Adam(1) - Alice(5)
Bob(2) - Barbara(5)
Charlie(3) - Elsie(4)
Dave(1) - Fiona(4)
Edgar(1) - Doris(5)
Fred(3) - Claire(6)
*** Execution Profile ***
Number of Choice Points: 19
Number of Failures: 12
Execution time: 7 msec

Test 'testCases-2' completed OK. Elapsed time 20.14 ms
```

It does not look like we received the best possible solutions. So, I decided to find ALL feasible solutions allowing people to select the best one themselves. To do this, I added the method ***solveAll()*** and used it inside the Excel table “SolveOptimizationProblem”*:*



```java
public Solution[] solveAll() {
    Solver solver = csp.getSolver();
    int numberOfSolutions = 0;
    SolutionIterator iter = solver.solutionIterator();
    ArrayList<Solution> solutions = new ArrayList<Solution>();
    while (iter.hasNext()) {
        numberOfSolutions++;
        Solution solution = iter.next();
        System.out.println("\nSolution #" + numberOfSolutions
                + " with total satisfied preferences: " + solution.getValue("All Preferences"));
        for (int i = 0; i < n; i++) {
            int value = solution.getValue("wife for " + men[i].getName());
            System.out.println(men[i].getName() + "(" + men[i].getPreferences()[value] + ") - "
                        + women[value].getName() + "(" + women[value].getPreferences()[i] + ")");
        }
        solutions.add(solution);
    }
    solver.logStats();
    return solutions.toArray(new Solution[solutions.size()]);
}
```

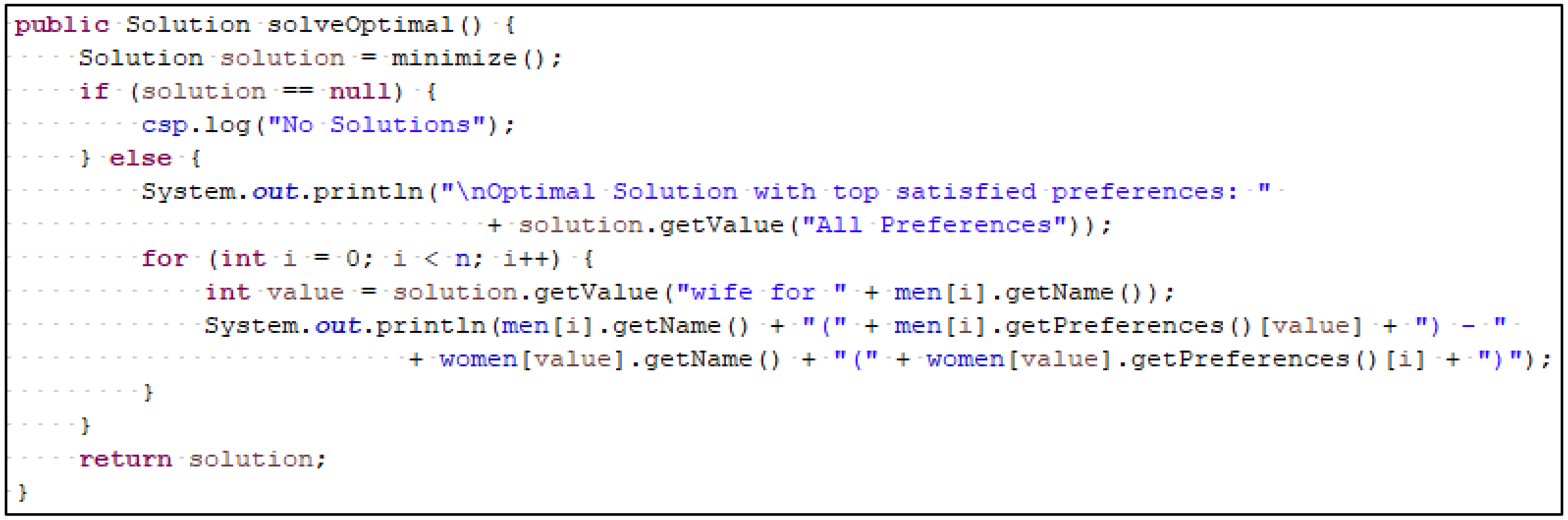
I received 5 feasible solutions for the Test Case 1:

Solution #1 All satisfied preferences: 25
Adam(1) - Barbara(3)
Bob(4) - Alice(2)
Charlie(2) - Claire(2)
Dave(3) - Doris(2)
Edgar(5) - Elsie(1)

```
Solution #2 All satisfied preferences: 24
Adam(1) - Barbara(3)
Bob(4) - Alice(2)
Charlie(1) - Elsie(3)
Dave(3) - Doris(2)
Edgar(2) - Claire(3)

Solution #3 All satisfied preferences: 24
Adam(1) - Barbara(3)
Bob(2) - Doris(4)
Charlie(2) - Claire(2)
Dave(1) - Alice(3)
Edgar(5) - Elsie(1)

Solution #4 All satisfied preferences: 23
Adam(1) - Barbara(3)
Bob(2) - Doris(4)
Charlie(1) - Elsie(3)
Dave(1) - Alice(3)
Edgar(2) - Claire(3)

Solution #5 All satisfied preferences: 26
Adam(4) - Doris(1)
Bob(4) - Alice(2)
Charlie(3) - Barbara(1)
Dave(4) - Claire(1)
Edgar(5) - Elsie(1)
*** Execution Profile ***
Number of Choice Points: 410
Number of Failures: 388
Execution time: 84 msec
Test 'testCases-1' completed OK. Elapsed time 159.95 ms
```

Note that Solution #4 is the same as our optimal solution from Approach 1.

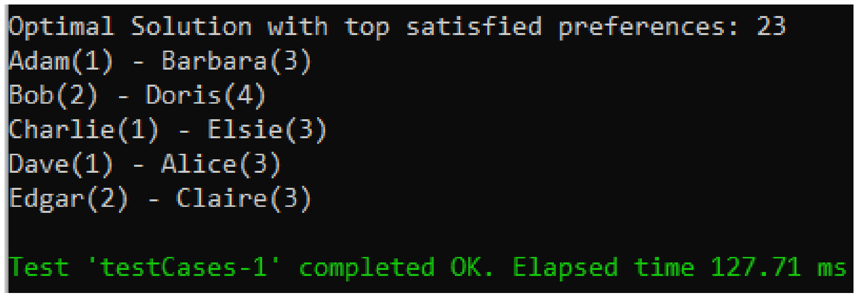
Then I decided to select the optimal solution among all these solutions. To do that, I added a new method *solveOptimal()* which I invoked in the Excel table “SolveOptimizationProblem” instead of *solveAll()*:



```
public Solution solveOptimal() {
    Solution solution = minimize();
    if (solution == null) {
        csp.log("No Solutions");
    } else {
        System.out.println("\nOptimal Solution with top satisfied preferences: "
                              + solution.getValue("All Preferences"));
        for (int i = 0; i < n; i++) {
            int value = solution.getValue("wife for " + men[i].getName());
            System.out.println(men[i].getName() + "(" + men[i].getPreferences()[value] + ") - "
                         + women[value].getName() + "(" + women[value].getPreferences()[i] + ")");
        }
    }
    return solution;
}
```

When I executed this decision model with solveOptimal, I received the following solutions):

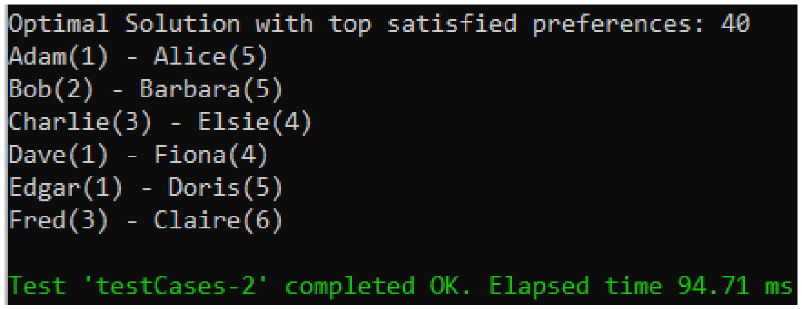
Test Case 1



```
Optimal Solution with top satisfied preferences: 23
Adam(1) - Barbara(3)
Bob(2) - Doris(4)
Charlie(1) - Elsie(3)
Dave(1) - Alice(3)
Edgar(2) - Claire(3)

Test 'testCases-1' completed OK. Elapsed time 127.71 ms
```

The same solution as in Approach 1

Test Case 2



```
Optimal Solution with top satisfied preferences: 40
Adam(1) - Alice(5)
Bob(2) - Barbara(5)
Charlie(3) - Elsie(4)
Dave(1) - Fiona(4)
Edgar(1) - Doris(5)
Fred(3) - Claire(6)

Test 'testCases-2' completed OK. Elapsed time 94.71 ms
```

This solution is slightly worse than in Approach 1

Conclusion

The resulting decision model demonstrates that it is not always necessary to implement complex rules/constraints in the way they are presented in plain English descriptions. We frequently may get the same or better results by minimizing the accumulated violation of major rules and constraints. The created two different implementations with various flavors are interchangeable and could be applied separately or in combinations.

The decision model provides a flexible infrastructure for representing and solving this relatively complex problem and its possible extensions. For instance, to make the decision model even more realistic we can relax marriage stability constraints and apply them only when spouse preferences are smaller than or equal to a certain "Stability Threshold". The decision model allows us to add more factors that will affect marriage stability without serious changes in the implementation. What is important is the fact that it is easy to add similar advancements to the decision model. What is important is the fact that it is easy to add similar advancements to the decision model.

The integrated use of rule engines and constraint solvers within the same business decision models not only simplifies their implementations. It also provides other benefits: the simplicity of adding more test cases (in Excel or JSON formats), standard one-button deployment of this decision service as AWS Lambda or MS Azure functions, and natural incorporation of developed decision services into any production power decision-making application.