Rule Violations and Over-Constrained Problems

Jacob Feldman, PhD
OpenRules Inc., CTO
jacobfeldman@openrules.com
www.openrules.com
www.4c.ucc.ie

Introduction

- Online decision support applications have to deal with situations when business rules are violated or contradict to each other. Why?
- It frequently happens not because of a bad rules design but because real-world problems are usually *over-constrained.* Frequent changes in the state of the system often violate the established rules
- To deal with rules violations we may apply techniques used in Constraint Programming (CP) for constraint violations. In this presentation I will:
  - Introduce concepts of **hard** and **soft** rules (constraints)
  - Share experience of measuring, controlling, and minimizing rule violations
  - Provide concrete examples of real-world problems along with Java code that shows different practical approaches of handling rule violations

Rules Violations

- Probably an ability to be violated is among the most important attributes of any rule
- Even the rules that never can be violated theoretically, are frequently violated in the real life
- Example:
  - Rule: “A person cannot be in two different places at the same time”
  - It should always be true but..
  - *Here is a real-life scheduling situation:*
    - *A human scheduler sends a worker to do two jobs in two different places after lunch*
    - *Job1 officially takes 2 hours*
    - *Job2 officially takes 3 hours*
    - *A worker has to do both jobs in 4 hours and… he will do it!*

Trivial example of over-constrained problem

- Variables:

  $x1 \in \{1,2\},\ x2 \in \{2,3\},\ x3 \in \{2,3\}$

- Constraints:

  $x1 > x2$ (i)

  $x1 + x2 = x3$ (ii)

- Solutions:

  There is no solution.

  Which is hardly useful in practice.

Some non-solutions might be regarded as reasonable

- Variables:

  x1 ∈ {1,2}, x2 ∈ {2,3}, x3 ∈ {2,3}

- Constraints:

  x1 > x2 (i)

  x1 + x2 = x3 (ii)

- Non- Solutions:

x_1	x_2	x_3	comment
1	2	2	all constraints violated
1	2	3	**first constraint violated only (minimum violation)**
1	3	2	all constraints violated
1	3	3	all constraints violated
2	2	2	all constraints violated
2	2	3	all constraints violated
2	3	2	all constraints violated
2	3	3	all constraints violated

- Usually business(!) specialists decide which “non-solution” is better

Example of a measured constraint violation

- Example:

  $x \in [9000,10000]$

  $y \in [0,20000]$

  $x \leq y$

- Let's "soften" the constraint '$x \leq y$' by introducing a cost variable z that represents the amount of violation as the gap between x and y.

- Suppose that we impose $z \in [0,5]$. By looking at the bounds of x and y, we can immediately deduce that $y \in [8995,20000]$

From examples to real life

- These trivial examples can be easily transferred to real-life problems
- **Loan Origination**:
  - A rules-based loan origination system rejects a student application for $30K education loan
  - Instead it could relax its hard rules and to offer a smaller loan of $28.5K to the same student
- **Job Scheduling** ("2 + 3 ~= 4"):
  - A strict automatic scheduler would not schedule Job1 (2 hours) and Job2 (3 hours) for a 4-hours shift
  - However, a human scheduler will! Why? A human knows the context: actual, not average job durations; capabilities of particular workers, etc.
  - To soften the hard rule ("2+3=5") introduce a concept of the "unavailability tolerance"

Constraint Satisfaction Problem - CSP

- A typical CSP problem deals with:
  - A set of constrained variables $X = \{ X_1, \ldots, X_n \}$
  - Each variable X_i is defined on a set of values $V_j = \{ v_{j1}, \ldots, v_{jk} \}$
  - A set of constraints $C = \{ C_1, \ldots, C_r \}$, where a constraint C_i defines relationships between different variables X_i by specifying the allowed combinations of their values
- To solve the CSP we usually need to find assignments of values to all variables that satisfies all the constraints. A set of such assignments called “solution”.

Over-Constrained Problems

- A problem is over-constrained if it has no solution, or in the above terms we cannot find a combination of values that satisfies all constraints
- In this situation, it is necessary to relax the problem constraints in order to find a solution
- The goal is to find a good compromise ("acceptable solution"), that is a total assignment of values that best respects the set of constraints, even if some of them are violated
- How to find an "acceptable solution"?
  - Soften (some of) the constraints of the problem
  - Compute solution that minimizes conflicts or maximizes satisfaction

A rules-based problem is similar to a CSP

- A typical rules-based problem deals with:
  - A set of business objects $X = \{ X_1, \ldots, X_n \}$
  - Each object X_i has properties $P_i = \{ p_1, \ldots, p_p \}$ with possible values $V_j = \{ v_{j1}, \ldots, v_{jk} \}$ for each property p_j
  - A set of rules $R = \{ R_1, \ldots, R_r \}$, where a rule R_i defines relationships between different objects/properties by specifying the allowed combinations of values for the properties
- To solve the problem we usually need to find assignments of values to all properties that satisfies all the rules. A set of such assignments called “solution”.

Example

- Rule:
  - If a driver with age<20 drives a sport car, then increase the premium by 2%
- Objects/Properties/Values:
  - **Driver** with the property “age” and values: 16 …100
  - **Car** with the property “type” and values: “sport”, “sedan”..
  - **Policy** with the property “premium”
- An increased premium can lead to a contradiction with another rule that limits the total premium

Hard and Soft Rules/Constraints

- A Hard Rule must always be satisfied
- A Soft Rule does not need to be satisfied, but we would like it to be
- How to decide which rules in a set of rules should be softened and which rules should remain hard?
- How to compare and control a degree of rule violations?

Using CP Experience

- Contrary to the BR approach, Constraint Programming (CP) has an extensive experience in dealing with real-life over-constrained problems
- The major CP approaches fall in one of two categories:
  - Quantitative approaches:

    Specify constraint violation costs and try to optimize an aggregated function defined on all cost variables
  - Qualitative approaches:

    Find explanations of conflicts and then recommend a preferred relaxation. There are promising research results for automatic conflict explanations and problem reformulation

Some Constraint Programming Approaches to over-constrained CSP

- Partial CSP [Freuder & Wallace, 1992]
  - maximize number of satisfied constraints
- Weighted CSP [Larossa, 2002]
  - associate a weight to each constraint
  - maximize weighted sum of satisfied constraints
- Possibilistic CSP [Schiex, 1992]
  - associate weight to each constraint representing its importance
  - hierarchical satisfaction of most important constraints, i.e. maximize smallest weight over all satisfied constraints
- Fuzzy CSP [Dubois et al., 1993] [Ruttkay, 1994]
  - associate weight to every tuple of every constraint
  - maximize smallest preference level (over all constraints)

CP "knows" how to soften many popular global constraints such as AllDiff

From Soft Constraints to Hard Optimization Constraints

- Cost-based approach ("Meta-Constraints on Violations for Over Constrained Problems" by T.Petit, J-C.Regin, C. Bessiere , 2000)
  - Introduce a cost variable that represents a violation measure of the constraint
  - Post meta-constraints on the cost variables
  - Optimize aggregation of all cost variables (e.g., take their sum, or max)
- Soft constraints become hard optimization constraints
  - Soft Constraint Satisfaction Problem is reformulated into Hard Constraint Optimization Problem
  - Cost variables can be used in other meta-constraints!

    if ($z_1 > 0$) then ($z_2 = 0$)

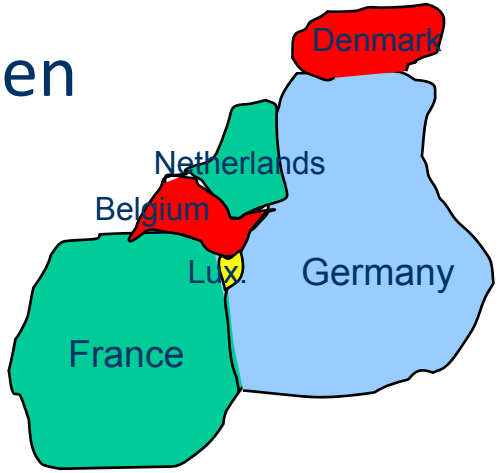
    if a nurse worked extra hours in the evening she cannot work next morning
- Now we can apply classical constraint programming solvers

Examples of Over-Constrained Problems

- We will demonstrate concrete methods of dealing with over-constrained problems using the following examples:
  - Map Coloring
    - with insufficient number of colors
  - Financial Portfolio Balancing
    - keeping a portfolio as close as possible to a “optimal” portfolio
    - while some stock allocation rules being violated
  - Miss Manners
    - with data that violates gender rules
    - with data that violates hobby rules
    - both

CSP Sample: "Map Coloring"

- A map-coloring problem involves choosing colors for the countries on a map in such a way that at most 4 colors are used and no two neighboring countries have the same color
- We will consider six countries: Belgium, Denmark, France, Germany, Netherlands, and Luxembourg
- The colors are blue, white, red or green



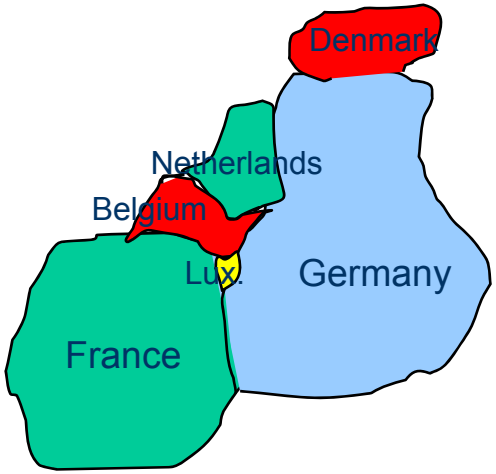
Example "Map Coloring": problem variables

```
static final int MAX = 4;  // number of colors

CSP p = new CSP();

// Constrained Variables
Var Belgium     =  p.addVar("Belgium",     0, MAX - 1);
Var Denmark     =  p.addVar("Denmark",     0, MAX - 1);
Var France      =  p.addVar("France",      0, MAX - 1);
Var Germany     =  p.addVar("Germany",     0, MAX - 1);
Var Netherlands = p.addVar("Netherlands",0, MAX - 1);
Var Luxemburg   =  p.addVar("Luxemburg",   0, MAX - 1);
```

Each country is represented as a variable that corresponds to an unknown color: 0,1,2, or 3

"Map Coloring": problem constraints

```
// Define Constraints
France.ne(Belgium).post();
France.ne(Luxemburg).post();
France.ne(Germany).post();
Luxemburg.ne(Germany).post();
Luxemburg.ne(Belgium).post();
Belgium.ne(Netherlands).post();
Germany.ne(Netherlands).post();
Germany.ne(Denmark).post();
```



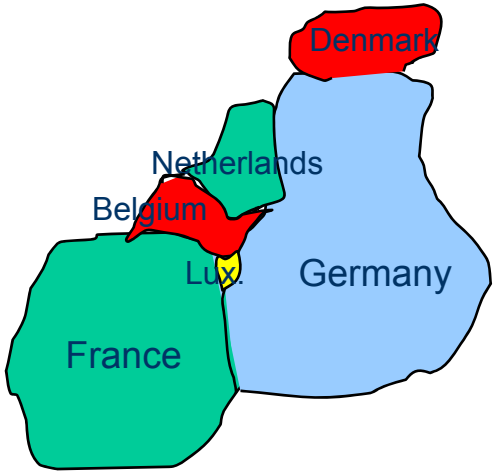

```
// We actually create a "not-equal" constraint and
then post it
Constraint c = Germany.ne(Denmark);
c.post();
```

"Map Coloring": solution search

```
// Solve
Solution solution = p.solve();
if (solution != null)  {
   for (int i = 0;  i < p.getVars().length;  i++)  {
        Var var = p.getvars()[i];
        p.log(var.getName() + " - " + colors[var.getValue()]);
   }
}
```

// Solution:

Belgium – red
Denmark – red
France – green
Germany – blue
Netherlands – green
Luxemburg - yellow



Over-Constrained Map Coloring Problem

- **Consider a map coloring problem when there are no enough colors, e.g. only two colors** (MAX=2)
- **Constraint "softening" rules:**
  - Coloring violations may have different importance on the scale 0-1000:
    - Luxemburg– Germany (9043)
    - France – Luxemburg (257)
    - Luxemburg – Belgium (568)
- **We want to find a solution that minimizes the total constraint violation**

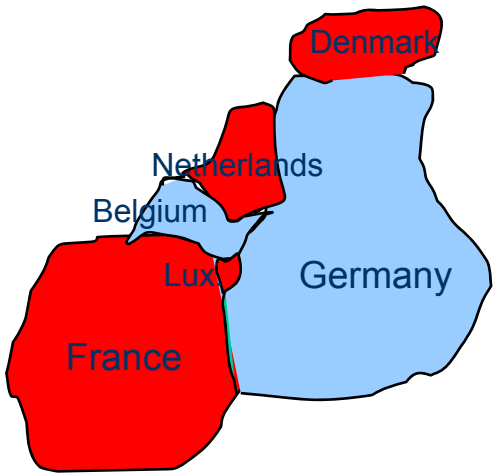
Map Coloring with Hard and Soft Constraints

```
// Hard Constraints
France.ne(Belgium).post();
France.ne(Germany).post();
Belgium.ne(Netherlands).post();
Germany.ne(Denmark).post();
Germany.ne(Netherlands).post();
// Soft Constraints
Var[] weights = {
      France.eq(Luxemburg).toVar().mul(257),
      Luxemburg.eq(Germany).toVar().mul(9043),
      Luxemburg.eq(Belgium).toVar().mul(568)
};
Var weightedSum = p.sum(weights);
```

22

Minimize Total Constraint Violations

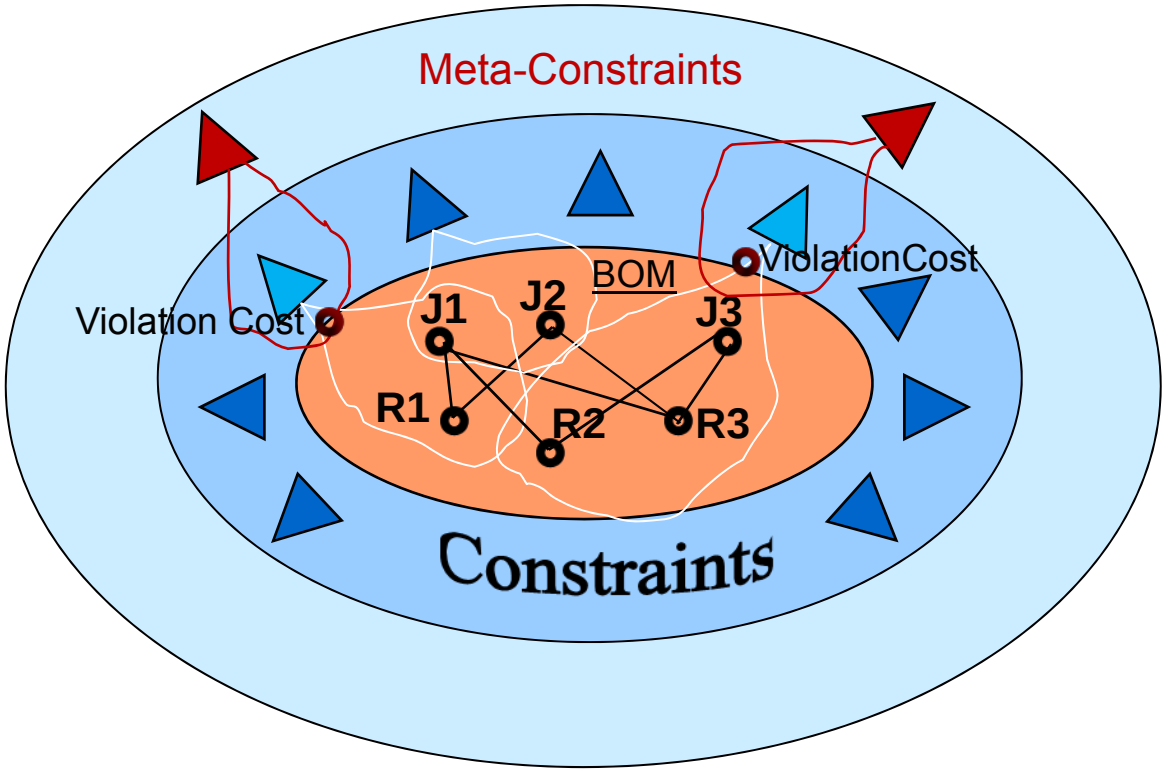
```
// Optimal Solution Search

Solution solution =  p.minimize(weightedSum);

if (solution == null)

        p.log("No solutions found");

else

        solution.log();
```



Solution:

Belgium[0] Denmark[1] France[1] Germany[0] Netherlands[1] Luxemburg[1]

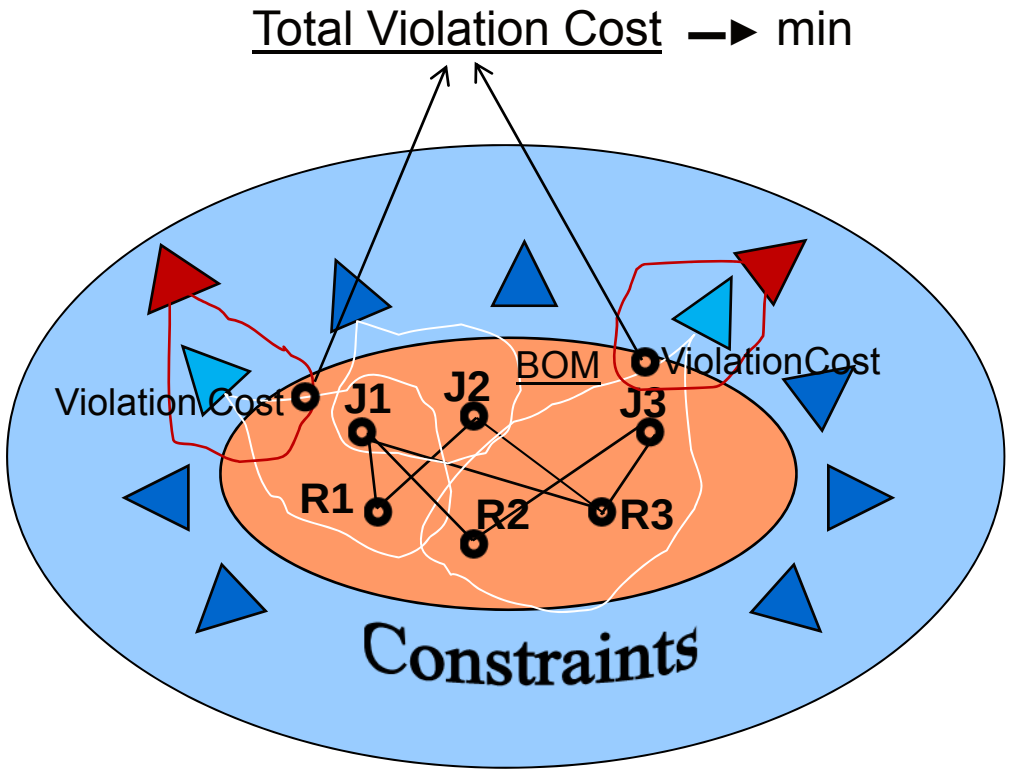
Adding Meta Constraints for Soft Constraints



- The meta-constraints are rules about violation costs for soft constraints

CSP with Soft Constraints => Optimization Problem

- "Meta-constraints" effectively convert a problem with soft constraints to a classical "hard" optimization problem



"Meta-constraints" are business rules

- Definitions of constraint violation costs and related "meta-constraints" do not actually belong to constraint programming
- These "meta-constraints" are usually defined by subject-matter experts (not programmers!) and thus can be expressed in business rules
- So, it is only natural to integrate BR and CP in a such way when:
  - BR define a softened CSP problem (or sub-problems)
  - CP solves the optimization problem

Example "Balancing Financial Portfolio"

- The "target" portfolio is defined as an active set of rules that describes a shape of every particular portfolio
- Rules Violations:
  - Permanent fluctuations of stock prices forces stock allocation rules to be "a little bit" violated at any time
- Objective:
  - Keep all portfolios as close as possible to the current "target" portfolio

Example: Portfolio Management Rules (hard)

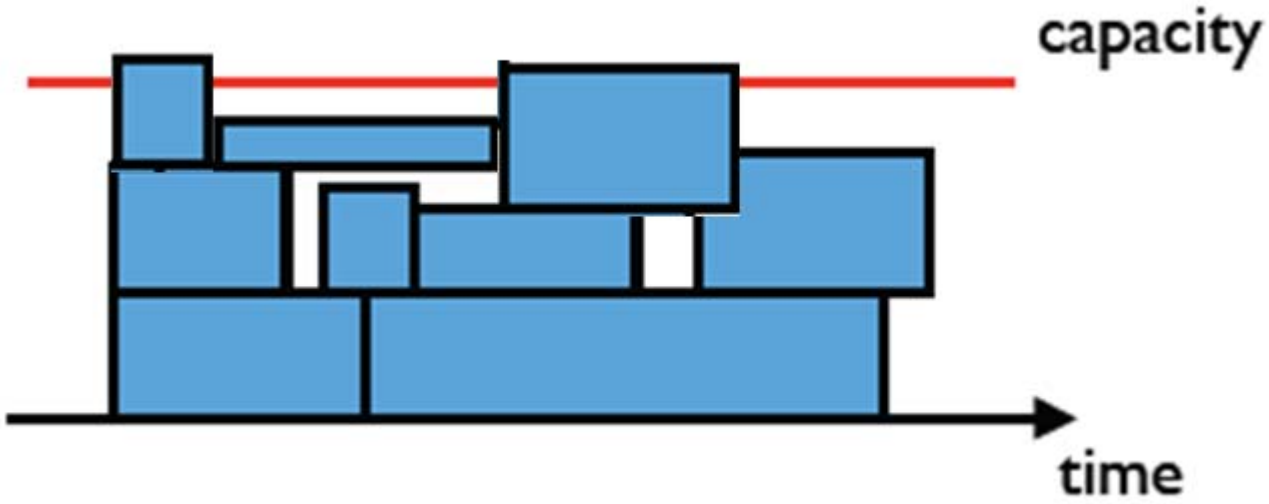
Rules void allocationRules(Portfolio portfolio)		
IF Selection Criteria	THEN Set Allocation Percent	
	Min	Max
Financial Sector	18	24
Utilities	19	25
Technology Sector	13	17
Retail Sector	6	
Pharmaceutical Sector	7	15
European except UK		10
Cash	5	

Softening the Portfolio Management Rules

Rules void allocationRules(Portfolio portfolio)						
IF Selection Criteria	THEN Set Allocation Percent		Set Rule Properties			
	Min	Max	Hard/Soft	Importance	Maximal Violation	Possible to Exceed
Financial Sector	18	24	Soft	8	1	
Utilities	19	25	Soft	7	0.5	
Technology Sector	13	17	Soft	9	3	Yes
Retail Sector	6		Hard			
Pharmaceutical Sector	7	15	Soft	5	2	
European except UK		10	Soft	6	4	Yes
Cash	5		Hard			

Typical Scheduling Constraints

- Given set of activities, each with processing time, resource consumption, earliest start time and latest end time, assign an execution time to each activity so that a given resource does not exceed its capacity ("global cumulative constraint")

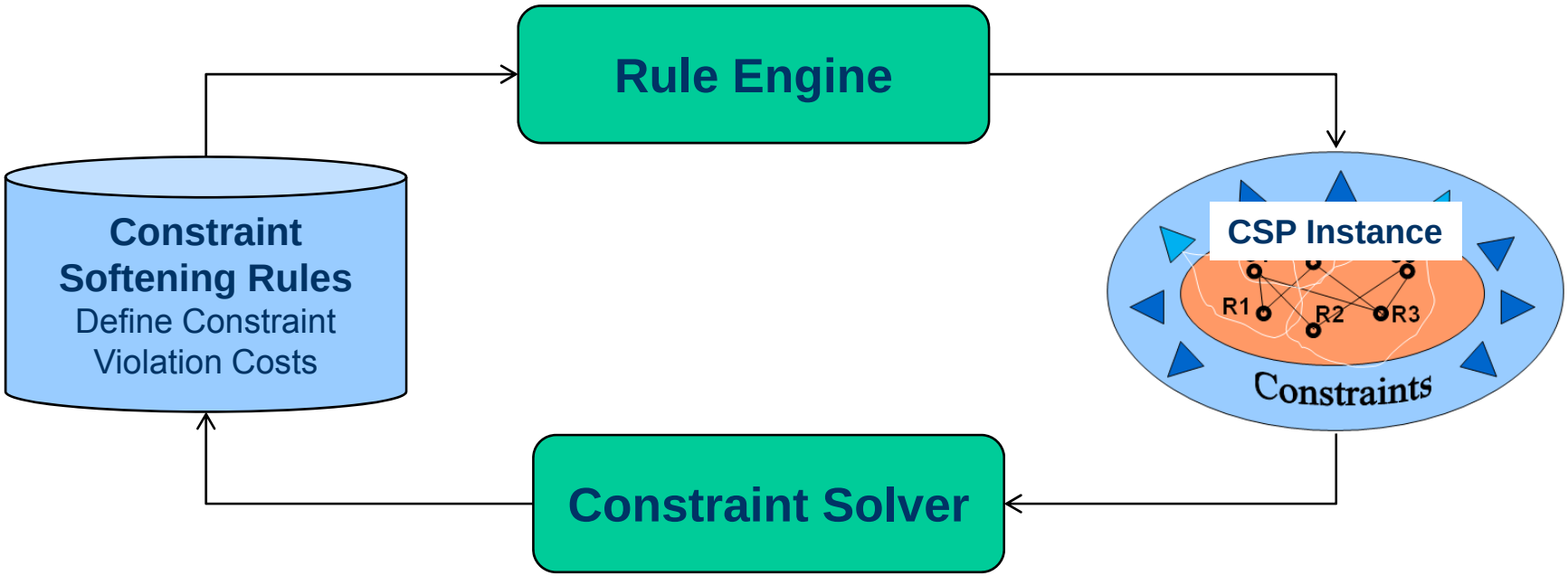


Softening Scheduling Constraints

- **Violation measures:**
  - Number of late activities
  - Acceptable overcapacity of resource during different time periods
  - Use of overtime
  - Overuse of skills
  - Assigning weights to worker preferences
- **Finding a compromise between conflicting scheduling objectives**
  - Minimize travel time
  - Start jobs ASAP
  - Minimize overtime
  - Maximize skills utilization
  - Satisfy worker preferences as much as possible

Constraint Softening Rules

Rule Engine and Constraint Solver working together



An Integrated Approach - Who Does What

BR Engine defines a business problem and generates a related CSP

Business User

CP Solver validates if the problem is over-constrained and points to inconsistencies/conflicts

CP Expert

BR Engine reformulates/softens the problem by defining "constraint softening rules" and generates a new CSP

Business User

CP solver (along with LP/MIP solvers) solves a reformulated optimization problem

CP Expert

Returns the results to the rule engine for further analysis

Miss Manners

- The "Miss Manners":
  - Find an acceptable seating arrangement for guests at a dinner party
  - Match people with the same hobbies (at least one), and to seat everyone next to a member of the opposite sex
- A popular benchmark for Rete-based rule engines
- We will consider situations when gender or/and hobby rules cannot be satisfied
  - First we will provide a CP-based solution for the benchmark data sets (that are well-tuned)
  - Then we will try to solve *over-constrained Miss Manners* when:
    - we do not have the same number males and females but still want to place them in the table "proportionally" (soft gender constraint)
    - Not all people have common hobbies (soft hobby constraints)

Miss Manners with hard constraints – define Guest

```
public class Guest {   // simple Java bean
        String name;
        String gender;
        int[] hobbies;
        int seat;  //result
        …
}
```

Miss Manners with hard constraints – a test

```
public class MannersSolver {
    Guest[] guests;
    CSP csp;
    Var[] guestVars;
    Var[] genderVars;
    VarSet[] hobbyVars;

    …

    public static void main(String[] args)  {
      OpenRulesEngine engine = new OpenRulesEngine("file:rules/Manners.xls");
      Guest[] guests = (Guest[]) engine.run("getGuests128”);
      MannersSolver solver = new MannersSolver(guests);
      solver.define();
      solver.solve();
    }
```

Miss Manners with hard constraints – define problem

```
public void define() {
  try {
      int N = guests.length;
// Prepare an array of N genders and an array of N hobby sets
     int[] guestGenders = new int[N];
     Set<Integer>[] hobbySets = new Set[N];
     for (int i = 0; i < N; i++) {
         Guest g = guests[i];
         guestGenders[i] = g.getGender().equals("m")? 0 : 1;
         hobbySets[i] = new HashSet<Integer>();
         for (int j = 0; j < g.getHobbies().length; j++) {
              hobbySets[i].add(new Integer(g.getHobbies()[j]));
         }
     }
     // Define CSP …..
  }
  catch (Exception e) {
    System.out.println("Problem is over-constrained");
  }
```

Miss Manners with hard constraints – define unknowns

```
...
csp = new CSP("Manners");
// Define problem variables for guest's seats
 guestVars = new Var[N];
 genderVars = new Var[N];
 hobbyVars = new VarSet[N];
 for (int i = 0; i < N; i++) {
     guestVars[i] = csp.addVar("guest-" + guests[i].getName(), 0, N - 1);
     genderVars[i] = csp.elementAt(guestGenders, guestVars[i]);
     genderVars[i].setName("gender-" + (i+1));
     hobbyVars[i] = csp.elementAt(hobbySets, guestVars[i]);
     hobbyVars[i].setName("hobbies-" + (i+1));
 }
```

Miss Manners with hard constraints – define constraints

```
// Post constraint "All guests occupy different seats“
csp.allDiff(guestVars).post();
// Post Gender and Hobby Constraints
for (int i = 0; i < N; i++) {
      Var currentGender = genderVars[i];        Var nextGender;
      VarSet currentHobbySet = hobbyVars[i];  VarSet nextHobbySet;
      if (i == N - 1) {
            nextGender = genderVars[0];   nextHobbySet = hobbyVars[0];
      }
      else {
           nextGender = genderVars[i+1]; nextHobbySet = hobbyVars[i+1];
      }
      // post constraints: Seat everyone next to a member of the opposite sex
      currentGender.ne(nextGender).post();
      // post constraints: Match people with the common hobbies
      currentHobbySet.createIntersectionWith(nextHobbySet).setEmpty(false);
}
```

Miss Manners with hard constraints – solve

```
public boolean solve() {
   // Find One Solution
   Goal goal1 = csp.goalGenerate(guestVars);
   Goal goal2 = csp.goalGenerate(hobbyVars);
   Solution solution = csp.solve(goal1.and(goal2));
   if (solution == null) {
      csp.log("No Solutions");
      return false;
   }
   else {
      for (int i = 0; i < N; i++) {
         Guest guest = guests[guestVars[i].getValue()];
         guest.setSeat(i+1);
         csp.log("Seat " + (i+1) + ": " + guest);
      }
      return true;
   }
}
```

Miss Manners with hard constraints – Solutions

Seat 1: Guest 1	Gender=m	Hobbies: 2 1 3
Seat 2: Guest 2	Gender=f	Hobbies: 2 1 3
Seat 3: Guest 3	Gender=m	Hobbies: 3 2
Seat 4: Guest 7	Gender=f	Hobbies: 1 2 3
Seat 5: Guest 4	Gender=m	Hobbies: 3 2 1
Seat 6: Guest 11	Gender=f	Hobbies: 1 3 2
Seat 7: Guest 5	Gender=m	Hobbies: 2 1 3
Seat 8: Guest 12	Gender=f	Hobbies: 3 1 2
Seat 9: Guest 6	Gender=m	Hobbies: 2 3 1
Seat 10: Guest 13	Gender=f	Hobbies: 2 3
Seat 11: Guest 8	Gender=m	Hobbies: 3 1
Seat 12: Guest 14	Gender=f	Hobbies: 1 2
Seat 13: Guest 9	Gender=m	Hobbies: 2 3 1
Seat 14: Guest 15	Gender=f	Hobbies: 2 3 1
Seat 15: Guest 10	Gender=m	Hobbies: 3 2 1
Seat 16: Guest 16	Gender=f	Hobbies: 2 3

Execution time: 63 msec

Problem Size	Execution Time Hard Constraints (milliseconds)
16	63
32	109
64	250
128	281

Finding all solutions

- By changing the standard search method "**solve**" to "**solveAll**" we may find all possible solutions
- I limited the search to 50 solutions and was able to find all of them in no time
- However, remember that our data was well-tuned.
- Back to the real-world…

Data change "kills" the optimism

- What if we change a gender for just 1 guest? (see Manners.xls)
- The problem will become over-constrained. Can the same code prove that there are no solutions?
- NO! It runs forever (at least > 20 mins even for 16 guests)
- What to do?
- Of course we can pre-process data to find this particular conflict. But will we be able to find other possible conflicts in data?
- You may try to run a pure rule engine benchmark against "bad" data..
- I decided first to look at possible symmetries…

Breaking Symmetries

- CP people know that when a search takes too much time try to get rid of symmetrical solutions
- In our case, a symmetry occurs when we switch guests that have the same gender and hobbies
- To break this type of symmetry I did the following:
  - Defined groups of guests that have the same gender and hobbies and specified for each *guest[i]* a constrained variable *seatVar[i]*
  - For all guests inside each group I posted additional symmetry breaking constraints:

    seatVars[i] < seatVars[i+1]

Breaking Symmetries helps.. but only partially

- By posting the symmetry breaking constraints the same program managed to prove that the problem with 16 guests does not have a solution
- It took ~20 seconds - compare with only 60 milliseconds when a solution was found
- However, the program fails to prove that the hard constraints are unsolvable for a larger number of guests such as 32, 64, and 128
- Conclusion: some hard constraints should become soft!

Softening Miss Manners Constraints

- The AllDiff constraint "**All guests occupy different seats**" remains hard
- We will soften the gender constraint "**Everyone should seat next to a member of the opposite sex**" that was actually violated in the "bad" data set
- To do this we may associate a violation cost 1 with every constraint violation:

  a seating .. f-m-m-f .. has a violation cost that is a constrained variable [0;1]

- Total cost violation is a sum of all cost violation variables
- Now, we reformulate our problem as an optimization problem:
  instead of finding a solution try to minimize the total cost violation!

Softening Miss Manners Constraints – code changes

- Define an array of violations: *Var[] violations:*

  Instead of the hard constraint:

  currentGender.ne(nextGender).post();

  we will define a violation variable for each pair of "current" and "next" genderVars:

  violations[i] = currentGender.eq(nextGender).toVar();

- Define the total violation:

  Var totalViolation = csp.sum(violations);

  totalViolation.ge(violationMin).post();

  Where: *int violationMin = Math.abs(males – females);*

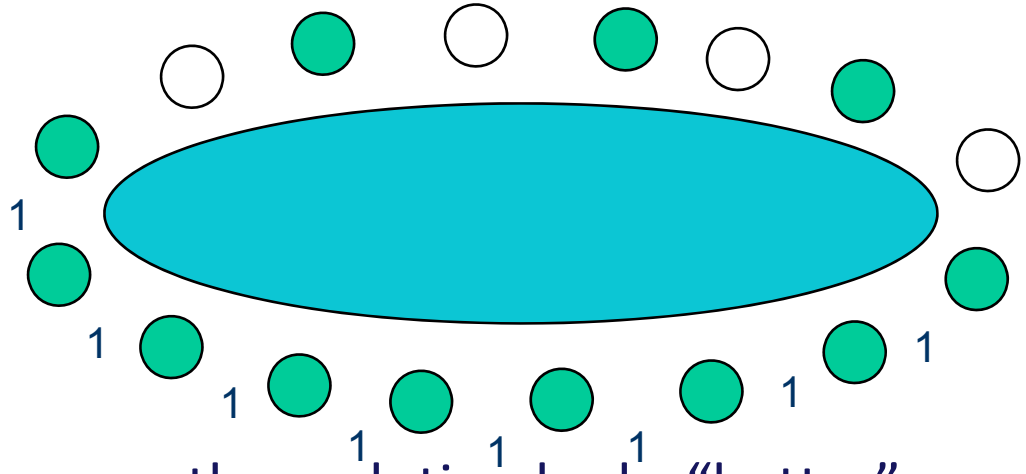
Soften problem is solved

- Applying this approach we successfully found optimal solutions for over-constrained Miss Manners problems of size 16 and 32

Problem Size	Execution Time (mills) Hard Gender Constraints (prove that there are no solutions)		Execution Time (mills) Soft Gender Constraints (Find a MaxCSP solution)			
	Symmetry Constraints OFF	Symmetry Constraints ON	Symmetry Constraints OFF		Symmetry Constraints ON	
			Time	Violations	Time	Violations
16	∞	19219	66984	4	8157	2
32	∞	∞	188 438	14 6	156 390	14 6

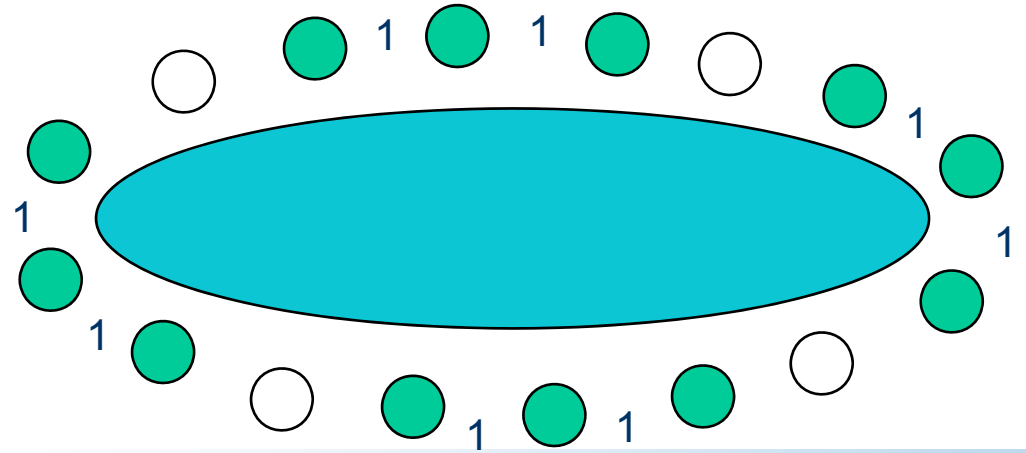
Solutions with minimal total constraint violations

- All constrain violations have the same cost 1. As a result, this solution has the minimal total constraint violation cost 8.



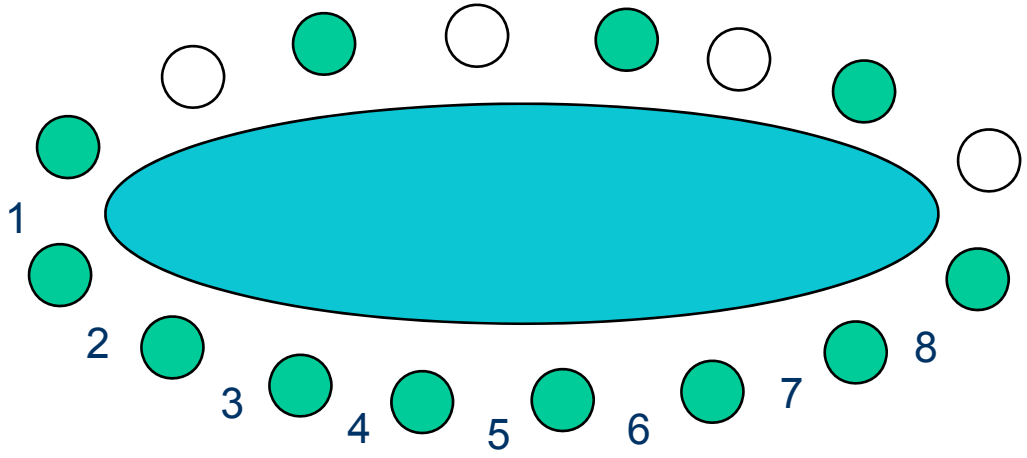
Total Violation Cost = 8

- However another solution looks “better”:

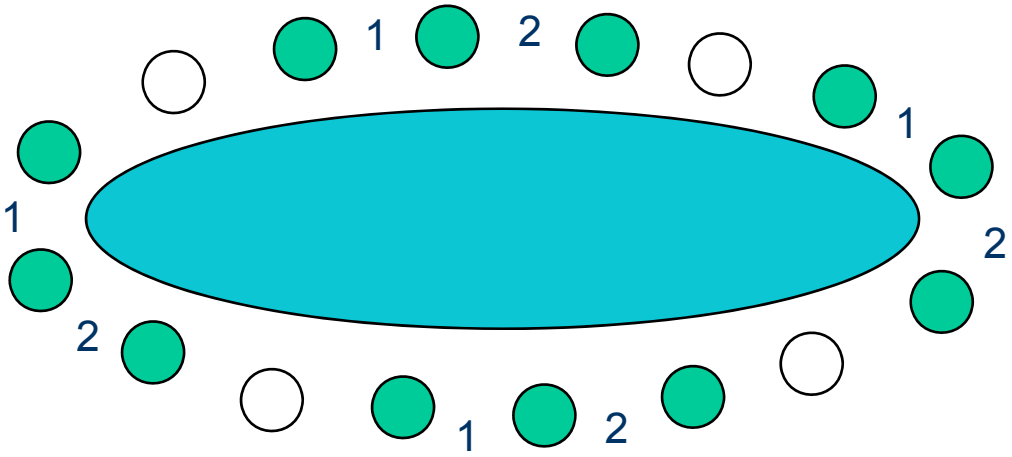


Total Violation Cost = 8

Assigning different costs to different constraint violations



Total Violation Cost = 36



Total Violation Cost = 12

The proper cost assignment code

- The array of constraint "*violations" consist of [0;1] variables*
- Now we may define an array of violation costs*:*

```
violations[0].eq(0).post();
Var[] violationCosts = new Var[N];
for (int i = 0; i < N; i++) {
      if (i==0)
            violationCosts[i] = violations[i];
      else {
            Var newCost = violationCosts[i-1].add(1);
            violationCosts[i] = violations[i].mul(newCost);
      }
}
totalViolation = csp.sum(violationCosts);
```

Integrating Hard and Soft Rules

- **We may suggest a more generic approach:**
  1. Try to solve a business problem with *hard constraints* first
  2. If after a certain time limit, a solution is not found, consider the problem to be over-constrained
  3. Try to solve the optimization problem with *softened constraints* setting time limits for:
     - Finding one solution with probably not-yet-optimal cost
     - Finding an optimal solution that minimizes the total constraint violations
- **This approach provides a practical compromise between time and quality**

Optimization Problem can be to hard to solve..

- Applying this approach we successfully found optimal solutions for over-constrained Miss Manners problems of size 16 and 32
- However, in practice a reformulated optimization problem could be too big to be solved with basic search strategies
  - For example, the same implementation failed to find optimal solutions for softened Miss Manners with a number of guests equals 64 or 128
- A more sophisticated search strategies may be required

Alternative Approaches

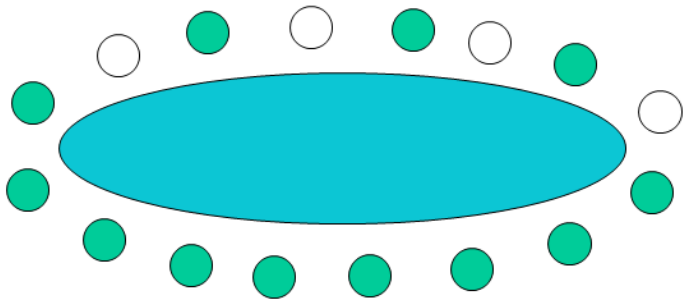
- More sophisticated optimization methods
  - A hybrid of CP and LP solvers
  - Reformulation to a graph search problems
- How about a Greedy Search (that never backtracks)?
  1. Place the first guest to the first seat
  2. Find a guest to be seated on the next seat with respect to gender and hobby rules:
     - We may give all not-yet-seated guests a score based on the relative scores: 10 points for a “good” gender, 1 point for each “good” hobby
     - A guest with the highest score will be selected
  3. If there are empty seats, go to 2.
- The Greedy Search can be implemented with rules or with constraints, but it can be easily implemented with a pure Java!

A pure Java solution (1)

```
public class MannersSolverGreedy {
    Guest[] guests;
    static int GENDER_SCORE = 10;
    static int HOBBY_SCORE = 1;

    public MannersSolverGreedy(Guest[] guests) {
        this.guests = guests;
    }

    public static void main(String[] args) {
        OpenRulesEngine engine = new OpenRulesEngine("file:rules/Manners.xls");
        Guest[] guests = (Guest[]) engine.run("getGuests16bad");
         MannersSolverGreedy solver = new MannersSolverGreedy(guests);
         Guest prevGuest = null;
         for (int seat = 0; seat < guests.length; seat++) {
              Guest guest = solver.findGuestToSeatNextTo(prevGuest);
               guest.seat = seat;
               System.out.println("Seat " + (seat+1) + ": " + guest);
              prevGuest = guest;
         }
     }
```

A pure Java solution (2)

```
Guest findGuestToSeatNextTo(Guest prevGuest) {
    if (prevGuest==null)
        return guests[0];
    int maxScore = -1;
    Guest maxGuest = null;
    for (int g = 0; g < guests.length; g++) {
        if (guests[g].seat >= 0)
            continue;
        int score = 0;
        if (!guests[g].sameGenderAs(prevGuest))
            score += GENDER_SCORE;
        int commonHobbies = prevGuest.commonHobbiesWith(guests[g]);
        score += HOBBY_SCORE * commonHobbies;
        if (score > maxScore) {
            maxScore = score;
            maxGuest = guests[g];}
    }
    return maxGuest;
}
```



Oversimplified Java vs Rules/Constraints

- In real life a “greedy” Java solution is a serious competitor to BR/CP-based solutions
- If BR/CP approach looks too complex, a customer may choose a “homemade” pure Java solution in spite of its apparent drawbacks:
  - Procedural implementation (rules are hidden in a search procedure)
  - While initially a more complex rules/constraints still could be added, it becomes not maintainable after several iterations
  - It is always “partial” and not optimal!
  - When a number of inter-related rules/constraints is large enough, a procedural code becomes not only unreadable but hardly implementable
- To win, a more superior BR/CP solution still has to be efficient, simple to implement, to understand, and to maintain!

And Finally.. about CP Tools

- There many CP solvers (open source and commercial)
- CP Standardization
  - www.cpstandards.com
  - JSR-331 "Constraint Programming API"
- Integration between BR and CP tools
  - "Rule Solver" is an OpenRules integrated solution
  - Use Java CP API to add CP capabilities to your BR tools

Summary

- **Integration of BR and CP empowers decision making systems with practical tools that address over-constrained problems**
  - Allow BR to define which rules to soften and how to do it
  - Allow CP to solve a reformulated optimization problem
- **BR and CP decision services working together is a powerful way to find the best soften solutions for real-world over-constrained problems**

Thank you

Q & A

You may contact me at

jacobfeldman@openrules.com

or

j.feldman@4c.ucc.ie