Roberto Zanotto 443065

Final Term di Programmazione Avanzata (testo del 26/01/13)

Esercizio 1:

Per rappresentare i due diversi tipi di vincoli ho utilizzato le classi `EqConstraint` e `DiseqConstraint` (che stanno per equality e disequality constraint). Entrambe estendono la stessa superclasse `Constraint`, con campi `left` e `right`. Un vincolo (x_i, y_j) è rappresentato con un'istanza dell'appropriata sottoclasse di `Constraint` avente x_i come `left` e y_j come `right`.

File Constraint.java:

```
public class Constraint {

   public String left, right;

   public Constraint() {
   }

   public Constraint(String left, String right) {
      this.left = left;
      this.right = right;
   }
}
```

File EqConstraint.java:

```
public class EqConstraint extends Constraint {

   public EqConstraint() {
      super();
   }

   public EqConstraint(String left, String right) {
      super(left, right);
   }

   @Override public String toString() {
      return "("+left+", "+right+")";
   }
}
```

File DiseqConstraint.java:

```
public class DiseqConstraint extends Constraint {

   public DiseqConstraint() {
      super();
   }

   public DiseqConstraint(String left, String right) {
      super(left, right);
   }

   @Override public String toString() {
      return "!("+left+", "+right+")";
   }
}
```

Per le variabili ho usato la classe `Variable`, che comprende il nome della variabile stessa e il suo dominio. Per il dominio ho usato un `ArrayList<String>`. In generale quando ho avuto bisogno di arrays ridimensionabili ho usato la classe `ArrayList`, che è simile alla classe `Vector` per implementazione, ma è più leggera (in particolare non è thread-safe). Nell'esercizio 3 chiarirò perché la classe implementa l'interfaccia `Iterable<String>`, e darò la definizione dell'iteratore `VariableIterator`.

File Variable.java:

```
public class Variable implements Iterable<String> {

   public String name;
   public ArrayList<String> domain;

   public Variable(String name, ArrayList<String> domain) {
      this.name = name;
```

```
    this.domain = domain;
  }

  public VariableIterator iterator() {
    return new VariableIterator(this);
  }
}
```

La classe `Problem` contiene i campi `variables`, `eqConstraints` e `diseqConstraints` che rappresentano rispettivamente le variabili del problema e i suoi equality e disequality constraints. Ho aggiunto anche la funzione `findVariableFromValue`, che a partire da un determinato valore trova la variabile che contiene tale valore nel suo dominio (funzione che tornerà utile in seguito).

File Problem.java:

```
public class Problem {

  public ArrayList<Variable> variables;
  public ArrayList<EqConstraint> eqConstraints;
  public ArrayList<DiseqConstraint> diseqConstraints;

  public Problem(ArrayList<Variable> variables, ArrayList<EqConstraint>
  eqConstraints, ArrayList<DiseqConstraint> diseqConstraints) {
    // eventuale analisi statica qui
    this.variables = variables;
    this.eqConstraints = eqConstraints;
    this.diseqConstraints = diseqConstraints;
  }

  public Variable findVariableFromValue(String value) {
    for (Variable var: variables) {
      if (var.domain.contains(value))
        return var;
    }
    return null;
  }
}
```

Esercizio 2:

Lo Scanner è del tutto analogo a quello visto nelle slide del corso. L'unica differenza è che nel costruttore non chiamo le funzioni `resetSyntax`, `wordChars` e `ordinaryChar`, che in questo caso non è necessario chiamare. Infatti di default (se non viene chiamata la `resetSyntax`) lo `StreamTokenizer` di java è fatto per tokenizzare linguaggi C-like e riconosce già i token che servono per il nostro linguaggio (identificatori, parentesi, virgole ecc...).

File Scanner.java:

```
class Scanner {

  public enum TokenType { EQUAL, COMMA, NOT, L_PAREN, R_PAREN, L_BRACE,
  R_BRACE, IDE, EOF, INVALID_CHAR };

  private StreamTokenizer input;

  public Scanner(Reader reader) {
    input = new StreamTokenizer(reader);
  }

  public TokenType nextToken() throws IOException {
    switch (input.nextToken()) {
      case '=': return TokenType.EQUAL;
      case ',': return TokenType.COMMA;
      case '!': return TokenType.NOT;
      case '(': return TokenType.L_PAREN;
      case ')': return TokenType.R_PAREN;
      case '{': return TokenType.L_BRACE;
      case '}': return TokenType.R_BRACE;
      case StreamTokenizer.TT_WORD: return TokenType.IDE;
      case StreamTokenizer.TT_EOF: return TokenType.EOF;
      default: return TokenType.INVALID_CHAR;
    }
  }
```

```
  public String getString() {
    return input.sval;
  }
}
```

La grammatica di riferimento per il parsing è la seguente:

```
PROBLEM -> VARIABLES { CONSTRAINT_LIST } !{ CONSTRAINT_LIST }
VARIABLES -> VARIABLE [VARIABLES]
VARIABLE -> ide = { VALUES }
VALUES -> ide [, VALUES]
CONSTRAINT_LIST -> [CONSTRAINTS]
CONSTRAINTS -> CONSTRAINT [, CONSTRAINTS]
CONSTRAINT -> (ide, ide)
```

Per i due tipi di vincoli ho due sottoclassi diverse, ma dal punto di vista della sintassi i vincoli sono tutti uguali. Per evitare di riscrivere due volte le funzioni di parsing dei vincoli, che avrebbero differito solo per return type, ho usato i generics e meccanismi di reflection (il metodo `newInstance` della classe `Class`).

File Parser.java:

```
public class Parser {

  private Scanner scanner;
  private Scanner.TokenType currentToken;

  public Parser() {
  }

  private void nextToken() throws IOException {
    currentToken = scanner.nextToken();
  }

  private void expect(Scanner.TokenType token)
  throws SyntaxException, IOException {
    if (currentToken != token)
      throw new SyntaxException("expected token: "+token+", found:
      "+currentToken);
    nextToken();
  }

  // PROBLEM -> VARIABLES { CONSTRAINT_LIST } !{ CONSTRAINT_LIST }
  public Problem parseProblem(Reader reader) throws SyntaxException,
  IOException, InstantiationException, IllegalAccessException {
    scanner = new Scanner(reader);
    nextToken();

    ArrayList<Variable> vars = parseVariables();
    expect(Scanner.TokenType.L_BRACE);
    ArrayList<EqConstraint> eqConstraints =
      parseConstraints(EqConstraint.class);
    expect(Scanner.TokenType.R_BRACE);
    expect(Scanner.TokenType.NOT);
    expect(Scanner.TokenType.L_BRACE);
    ArrayList<DiseqConstraint> diseqConstraints =
      parseConstraints(DiseqConstraint.class);
    expect(Scanner.TokenType.R_BRACE);
    expect(Scanner.TokenType.EOF);

    return new Problem(vars, eqConstraints, diseqConstraints);
  }

  // VARIABLES -> VARIABLE [VARIABLES]
  private ArrayList<Variable> parseVariables()
  throws SyntaxException, IOException {
    ArrayList<Variable> vars = new ArrayList<Variable>();
    do {
      vars.add(parseVariable());
    } while (currentToken == Scanner.TokenType.IDE);

    return vars;
  }

  // VARIABLE -> ide = { VALUES }
```

```
  private Variable parseVariable() throws SyntaxException, IOException {
    String name = scanner.getString();
    expect(Scanner.TokenType.IDE);
    expect(Scanner.TokenType.EQUAL);
    expect(Scanner.TokenType.L_BRACE);
    ArrayList<String> domain = parseValues();
    expect(Scanner.TokenType.R_BRACE);

    return new Variable(name, domain);
  }

  // VALUES -> ide [, VALUES]
  private ArrayList<String> parseValues() throws SyntaxException, IOException {
    ArrayList<String> values = new ArrayList<String>();
    values.add(scanner.getString());
    expect(Scanner.TokenType.IDE);
    while (currentToken == Scanner.TokenType.COMMA) {
      nextToken();
      values.add(scanner.getString());
      expect(Scanner.TokenType.IDE);
    }
    return values;
  }

  // CONSTRAINT_LIST -> [CONSTRAINTS]
  // CONSTRAINTS -> CONSTRAINT [, CONSTRAINTS]
  private <C extends Constraint> ArrayList<C>
  parseConstraints(Class<C> constraintType) throws SyntaxException,
  IOException, InstantiationException, IllegalAccessException {
    ArrayList<C> constraints = new ArrayList<C>();
    if (currentToken == Scanner.TokenType.L_PAREN) {
      constraints.add(parseConstraint(constraintType));
      while (currentToken == Scanner.TokenType.COMMA) {
        nextToken();
        constraints.add(parseConstraint(constraintType));
      }
    }
    return constraints;
  }

  // CONSTRAINT -> (ide, ide)
  private <C extends Constraint> C parseConstraint(Class<C> constraintType)
  throws SyntaxException, IOException, InstantiationException,
  IllegalAccessException {
    expect(Scanner.TokenType.L_PAREN);
    String left = scanner.getString();
    expect(Scanner.TokenType.IDE);
    expect(Scanner.TokenType.COMMA);
    String right = scanner.getString();
    expect(Scanner.TokenType.IDE);
    expect(Scanner.TokenType.R_PAREN);

    C constraint = constraintType.newInstance();
    constraint.left = left;
    constraint.right = right;
    return constraint;
  }
}
```

File SyntaxException.java:

```
public class SyntaxException extends Exception {

  public SyntaxException() {
    super();
  }

  public SyntaxException(String message) {
    super(message);
  }
}
```

L'analisi statica, come per esempio il controllo dell'unicità degli identificatori, può essere eventualmente inserita nel costruttore della classe `Problem`.

Esercizio 3:

Per documentarmi sui vari metodi di risoluzione dei problemi CSP ho fatto riferimento al paper ***Algorithms and Heuristics for Constraint Satisfaction Problems***, in particolare al secondo capitolo (Algorithms from the Literature). Ho deciso di implementare un algoritmo di backtracking senza alcun tipo di forward checking (che avrebbe richiesto strutture dati ed algoritmi più sofisticati). Mostro innanzitutto la classe `VariableIterator`, che consente di iterare sui valori del dominio di una variabile. Si tratta di un semplice iteratore con in più i metodi `current` (che restituisce l'elemento corrente senza avanzare) e `reset` (che riporta l'iteratore all'inizio).

File VariableIterator.java:

```
class VariableIterator implements Iterator<String> {

   private Variable variable;
   private int currentValue;

   public VariableIterator(Variable variable) {
      this.variable = variable;
      currentValue = -1;
   }

   public String current() {
      return (currentValue == -1) ? null : variable.domain.get(currentValue);
   }

   public boolean hasNext() {
      return (currentValue+1) < variable.domain.size();
   }

   public String next() {
      if (!hasNext())
         throw new NoSuchElementException();
      currentValue++;
      return variable.domain.get(currentValue);
   }

   public void remove() {
      throw new UnsupportedOperationException();
   }

   public void reset() {
      currentValue = -1;
   }
}
```

L'algoritmo di ricerca della soluzione funziona nel modo seguente: considera la prima variabile non ancora assegnata (di indice `firstUnassignedVar`), sceglie un valore da assegnarle e controlla che tale valore sia consistente con le variabili già assegnate in precedenza (tramite la funzione `checkCompatibility`). Se tale valore risulta compatibile e se si riesce a trovare ricorsivamente un assegnamento per le altre variabili, si ha trovato una soluzione e `findSolutionRec` restituisce `true`. Altrimenti si va a provare con un altro possibile valore per la variabile considerata. Se nessun valore della variabile porta ad una soluzione, si restituisce `false` facendo di fatto backtracking. Il metodo pubblico che restituisce una soluzione (se esiste) è `findSolution`, che chiama `preprocessProblem` (spiegato in seguito) e `findSolutionRec` con `firstUnassignedVar = 0`. Il solver mantiene un array di iteratori (`varIterators`), che consente di iterare sui valori del dominio di ogni variabile e tiene inoltre traccia del valore correntemente assegnato ad ogni variabile.

File Solver.java:

```
public class Solver {

   protected Problem problem;
   protected VariableIterator[] varIterators;

   public Solver() {
   }

   public Solution findSolution(Problem problem) {
```

```
    this.problem = problem;
    preprocessProblem();
    return findSolutionRec(0) ? createSolutionFromVars() : null;
  }
  private boolean findSolutionRec(int firstUnassignedVar) {
    if (firstUnassignedVar >= varIterators.length)
      return true;
    VariableIterator varIterator = varIterators[firstUnassignedVar];
    while (varIterator.hasNext()) {
      String value = varIterator.next();
      if (checkCompatibility(firstUnassignedVar, value) && // short circuit!
        findSolutionRec(firstUnassignedVar+1))
          return true;
    }
    varIterator.reset();
    return false;
  }
  ...
}
```

Le soluzioni sono rappresentate dalla classe `Solution`, che contiene un assegnamento, ovvero una `HashMap` che associa ad ogni nome di variabile il suo valore. Il metodo del solver `createSolutionFromVars` si preoccupa solamente di creare un'appropriata istanza di `Solution`, una volta che è stata già trovata una soluzione del problema sotto forma di assegnamento opportuno delle variabili individuato da `varIterators`.

File Solution.java:

```
public class Solution {
  public HashMap<String, String> assignment;
  public Solution(HashMap<String, String> assignment) {
    this.assignment = assignment;
  }
}
```

File Solver.java:

```
public class Solver {
  ...
  protected Solution createSolutionFromVars() {
    HashMap<String, String> assignment =
      new HashMap<String, String>(varIterators.length);
    for (int i = 0; i < varIterators.length; i++) {
      assignment.put(problem.variables.get(i).name, varIterators[i].current());
    }
    return new Solution(assignment);
  }
  ...
}
```

Il costo di un algoritmo di backtracking è molto influenzato dal costo della funzione che abbiamo chiamato `checkCompatibility` (che infatti è l'unica funzione chiamata da `findSolutionRec`, oltre alla chiamata ricorsiva). Per questo motivo ho cercato di ottimizzare tale funzione il più possibile. Per fare ciò, prima di far partire l'algoritmo di solving, il problema viene “preprocessato” (`preprocessProblem`). La funzione `preprocessProblem` si occupa di due cose: inizializza l'array `varIterators` e modifica la rappresentazione dei vincoli del problema, salvandoli in una struttura dati che permette di velocizzare notevolmente la funzione `checkCompatibility`. Vogliamo salvare i vincoli del problema come coppie (in cui l'ordine non è significativo) di valori tra di loro incompatibili, che chiameremo coppie proibite. Un disequality constraint $!(x_i, y_j)$ significa “se $x = x_i$ allora deve valere $y != y_j$”, ovvero “non è valido un assegnamento che ha contemporaneamente $x = x_i$ e $y = y_i$”. Tale vincolo produrrà quindi la coppia proibita (x_i, y_j). Un equality constraint (x_i, y_j), invece, significa “se $x = x_i$ allora deve valere $y = y_j$”, ovvero “non è valido un assegnamento che ha contemporaneamente $x = x_i$ e $y = y_k$ se $y_k != y_j$”. Tale vincolo produrrà quindi le coppie proibite (x_i, y_k) per ogni $y_k != y_j$ nel dominio di y. Le coppie di valori sono rappresentate dalla classe `Pair` (con opportune funzioni `equals` e `hashCode`) e le coppie proibite vengono salvate nello `HashSet forbiddenPairs` del solver, in maniera da poter controllare in tempo O(1) se una determinata coppia di valori è o meno proibita.

File Solver.java:

```
public class Solver {
  ...
  private HashSet<Pair> forbiddenPairs;

  protected void preprocessProblem() {
    varIterators = new VariableIterator[problem.variables.size()];
    for (int i = 0; i < varIterators.length; i++)
      varIterators[i] = problem.variables.get(i).iterator();

    forbiddenPairs = new HashSet<Pair>(problem.eqConstraints.size()
    +problem.diseqConstraints.size());
    for (Constraint eqConstraint: problem.eqConstraints) {
      Variable var = problem.findVariableFromValue(eqConstraint.right);
      for (String value: var.domain) {
        if (!value.equals(eqConstraint.right))
          forbiddenPairs.add(new Pair(eqConstraint.left, value));
      }
    }
    for (Constraint diseqConstraint: problem.diseqConstraints) {
      forbiddenPairs.add(new Pair(diseqConstraint.left,
      diseqConstraint.right));
    }
  }
  ...
}
```

File Pair.java:

```
class Pair {

  private String a, b;

  public Pair(String a, String b) {
    if (a.compareTo(b) <= 0) {
      this.a = a;
      this.b = b;
    }
    else {
      this.a = b;
      this.b = a;
    }
  }

  @Override public boolean equals(Object o) {
    Pair p = (Pair) o;
    return p != null && a.equals(p.a) && b.equals(p.b);
  }

  @Override public int hashCode() {
    return (a+","+b).hashCode();
  }
}
```

Da notare che la classe `Pair` serve solamente al particolare algoritmo che abbiamo scelto per implementare il solver ed ha quindi senso che sia package-private.
La funzione `checkCompatibility` può essere così implementata in maniere semplice e con complessità lineare nel numero di variabili assegnate. Infatti, per controllare se l'ultimo assegnamento fatto (della variabile di indice `lastAssignedVar` con valore `lastValue`) è compatibile con i precedenti basta controllare se la coppia ([valore dell'`i`-esima variablie], `lastValue`) è una coppia proibita, per ogni `0 <= i < lastAssignedVar`.

File Solver.java:

```
public class Solver {
  ...
  protected boolean checkCompatibility(int lastAssignedVar, String lastValue) {
    for (int i = 0; i < lastAssignedVar; i++) {
      if (forbiddenPairs.contains(
      new Pair(varIterators[i].current(), lastValue)))
        return false;
    }
    return true;
```

```
    }
}
```

Esercizio 4:

Il generatore di soluzioni è rappresentato dalla classe `MultiSolver` che estende la calsse `Solver`. Ha un metodo `setProblem` e un metodo `nextSolution` che calcola (ogni volta che viene chiamato) la soluzione successiva al problema. Quando non ci sono più soluzioni restituisce `null`. L'algoritmo usato è lo stesso dell'esercizio 3 (si fa infatti grande riuso delle procedure dell'esercizio precedente), ma la ricorsione è stata sostituita con l'iterazione e il parametro `firstUnassignedVar` non è passato come argomento ma è salvato in un apposito campo del multisolver. Dopo aver individuato un determinato valore per la prima variabile non assegnata si incrementa `firstUnassignedVar` per andare a considerare la variabile successiva (dove prima facevamo la chiamata `findSolutionRec(firstUnassignedVar+1)`), mentre quando abbiamo esaurito i valori per una variabile facciamo backtracking decrementando `firstUnassignedVar` (dove prima restituivamo `false` al chiamante). Se arriviamo ad avere `fistUnassignedVar == varIterator.length` (il numero delle variabili) abbiamo assegnato tutte le variabili e abbiamo quindi trovato una soluzione; la restituiamo e decrementiamo `firstUnassignedVar` in maniera che l'algoritmo possa ripartire correttamente alla prossima chiamata di `nextSolution`. Se arriviamo ad avere `firstUnassignedVar == -1` abbiamo fatto backtracking sulla prima variabile e quindi non ci sono più soluzioni.

File MultiSolver.java:

```
public class MultiSolver extends Solver {

   private int firstUnassignedVar;

   public MultiSolver() {
      super();
   }

   public void setProblem(Problem problem) {
      this.problem = problem;
      preprocessProblem();
      firstUnassignedVar = 0;
   }

   public Solution nextSolution() {
      vars_loop:
      while (firstUnassignedVar >= 0 && firstUnassignedVar < varIterators.length)
      {
         VariableIterator varIterator = varIterators[firstUnassignedVar];
         while (varIterator.hasNext()) {
            String value = varIterator.next();
            if (checkCompatibility(firstUnassignedVar, value)) {
               firstUnassignedVar++;
               continue vars_loop;
            }
         }
         varIterator.reset();
         firstUnassignedVar--;
      }
      if (firstUnassignedVar < 0)
         return null;
      firstUnassignedVar--;
      return createSolutionFromVars();
   }

   @Override public Solution findSolution(Problem problem) {
      setProblem(problem);
      return nextSolution();
   }
}
```

Ho fatto l'override del metodo `findSolution` in maniera che restituisca comunque la prima soluzione del problema, ma usando le procedure proprie del multisolver, per evitare che una chiamata a `findSolution` in mezzo ad una serie di chiamate a `nextSolution` potesse lasciare lo stato del multisolver inconsistente; `Solver.findSolution` avrebbe infatti resettato `varIterators`, senza resettare `firstUnassignedVar`.

Esercizio 5:

Rappresento una spiegazione (`explanation`) di una soluzione come lista di vincoli che sono stati soddisfatti da quell'assegnamento e aggiorno quindi la classe `Solution` per contenere anche tale informazione:

File Solution.java:

```
public class Solution {

   public HashMap<String, String> assignment;
   public ArrayList<Constraint> explanation;

   public Solution(HashMap<String, String> assignment,
   ArrayList<Constraint> explanation) {
      this.assignment = assignment;
      this.explanation = explanation;
   }
}
```

Per calcolare la spiegazione per ogni soluzione è sufficiente modificare la funzione `createSolutionFormVars`. Infatti nel momento in cui `createSolutionFromVars` viene chiamata, conosciamo tramite `varIterators` tutti i valori che sono stati assegnati ad ogni variabile e anche l'ordine in cui i valori sono stati assegnati dal solver. Per verificare quali vincoli sono stati soddisfatti per arrivare alla soluzione controllo per il valore corrente di ogni variabile quali sono i vincoli che hanno come parte sinistra tale valore. Inoltre modificando `createSolutionFromVars` la nuova funzionalità sarà aggiunta sia al solver sia al multisolver.

File Solver.java:

```
public class Solver {
   ...
   protected Solution createSolutionFromVars() {
      HashMap<String, String> assignment =
        new HashMap<String, String>(varIterators.length);
      ArrayList<Constraint> explanation = new ArrayList<Constraint>();
      for (int i = 0; i < varIterators.length; i++) {
         String value = varIterators[i].current();
         assignment.put(problem.variables.get(i).name, value);

         for (EqConstraint eqConstraint: problem.eqConstraints) {
            if (eqConstraint.left.equals(value))
               explanation.add(eqConstraint);
         }
         for (DiseqConstraint diseqConstraint: problem.diseqConstraints) {
            if (diseqConstraint.left.equals(value))
               explanation.add(diseqConstraint);
         }
      }
      return new Solution(assignment, explanation);
   }
   ...
}
```

Esercizio 6:

Per il quadrato latino 3x3 ho usato 9 variabili, una per ogni casella della matrice 3x3, chiamate aa, ab, ac (prima riga), ba, bb, bc (seconda riga) e ca, cb, cc (terza riga). Ogni variabile ha dominio di cardinalità 3. Per esempio il dominio di aa è {aa1, aa2, aa3} e il valore aa2 indica che nella casella aa del quadrato ci sarà il valore 2. I vincoli sono tutti di tipo disequality constraint e fanno sì che non ci possa essere lo stesso valore in due caselle che stanno sulla stessa riga o colonna. Per esempio aa e ba stanno sulla stessa colonna, quindi avremo (tra gli altri) i vincoli !{(aa1, ba1), (aa2, ba2), (aa3, ba3)}. I vincoli per la prima riga sono quindi:

```
!{ (aa1, ab1), (aa1, ac1), (aa1, ba1), (aa1, ca1),
   (ab1, aa1), (ab1, ac1), (ab1, bb1), (ab1, cb1),
   (ac1, aa1), (ac1, ab1), (ac1, bc1), (ac1, cc1),

   (aa2, ab2), (aa2, ac2), (aa2, ba2), (aa2, ca2),
   (ab2, aa2), (ab2, ac2), (ab2, bb2), (ab2, cb2),
   (ac2, aa2), (ac2, ab2), (ac2, bc2), (ac2, cc2),
```

```
   (aa3, ab3), (aa3, ac3), (aa3, ba3), (aa3, ca3),
   (ab3, aa3), (ab3, ac3), (ab3, bb3), (ab3, cb3),
   (ac3, aa3), (ac3, ab3), (ac3, bc3), (ac3, cc3)  }
```

In realtà alcuni vincoli di quelli mostrati sono superflui. Per esempio i due vincoli !(aa1, ab1) e !(ab1, aa1) indicano entrambi la stessa cosa, ovvero che non si può avere contemporaneamente aa = aa1 e ab = ab1. Si può notare infatti che l'ordine dei valori in un generico disequality constraint non è significativo (cosa che non vale per gli equality constraints). Ciò deriva della proprietà (p => q) ≡ (!q => !p), per cui !(aa1, ab1) equivale a aa = aa1 => ab != ab1 che equivale a ab = ab1 => aa != aa1 che è la definizione di !(ab1, aa1). Eliminando i vincoli duplicati otteniamo quindi:

```
!{ (aa1, ab1), (aa1, ac1), (aa1, ba1), (aa1, ca1),
   (ab1, ac1), (ab1, bb1), (ab1, cb1),
   (ac1, bc1), (ac1, cc1),
   [analoghi per i valori 2 e 3] }
```

Faccio notare che l'algoritmo usato nel solver ignora automaticamente eventuali vincoli duplicati, perché essi producono le stesse coppie proibite (uguali secondo `Pair.equals`) e le coppie ripetute vengono ignorate con l'inserimento nello `HashSet forbiddenPairs`.
La soluzione trovata dal solver per il quadrato latino 3x3 è data dall'assegnamento:

```
{ aa=aa1, ab=ab2, ac=ac3,
  ba=ba2, bb=bb3, bc=bc1,
  ca=ca3, cb=cb1, cc=cc2  }
```

con `explanation`:

```
{ !(aa1, ab1), !(aa1, ac1), !(aa1, ba1), !(aa1, ca1),
  !(ab2, aa2), !(ab2, ac2), !(ab2, bb2), !(ab2, cb2),
  !(ac3, aa3), !(ac3, ab3), !(ac3, bc3), !(ac3, cc3),
  !(ba2, bb2), !(ba2, bc2), !(ba2, aa2), !(ba2, ca2),
  !(bb3, ba3), !(bb3, bc3), !(bb3, ab3), !(bb3, cb3),
  !(bc1, ba1), !(bc1, bb1), !(bc1, ac1), !(bc1, cc1),
  !(ca3, cb3), !(ca3, cc3), !(ca3, aa3), !(ca3, ba3),
  !(cb1, ca1), !(cb1, cc1), !(cb1, ab1), !(cb1, bb1),
  !(cc2, ca2), !(cc2, cb2), !(cc2, ac2), !(cc2, bc2)  }
```

I vincoli di `explanation` sono stati stampati in accordo al loro metodo `toString`.
Il multisolver trova correttamente tutte e 12 le soluzioni del problema.

Esercizio 7:

La nozione di reflection nell'ambito dei linguaggi di programmazione può essere definita come l'abilità di un programma di esaminare e modificare la struttura e il comportamento delle entità che lo compongono a runtime (A Tutorial on Behavioral Reflection and its Implementation). In java la reflection consente l'introspezione di classi, interfacce, campi e metodi a runtime, ovvero conoscendo il nome della classe/interfaccia/metodo da esaminare solo a runtime. Per fornire funzionalità di reflection è necessario che il compilatore di un linguaggio memorizzi informazioni sulle entità del linguaggio in strutture dati che dovranno essere disponibili a runtime (tipicamente interrogabili tramite l'API di reflection). Per esempio in C (che non ha reflection) informazioni come il nome e il tipo dei campi che costituiscono uno struct sono disponibili solamente a compile-time e si perdono con la compilazione (non vengono salvate nel file oggetto), mentre in java tali informazioni riguardanti i campi e metodi di una classe sono sono tenute in appropriate strutture dati salvate all'interno del bytecode, cosa che consente di esaminare la struttura delle classi a runtime. La reflection è disponibile tipicamente in linguaggi di alto livello. Esempi di linguaggi di programmazione con reflection sono: java, C#, Objective-C, ML e Haskell. La reflection è inoltre presente in numerosi linguaggi di scripting.
Se abbiamo un array di oggetti e vogliamo eseguire su ognuno di essi un'operazione diversa a seconda del loro tipo attuale si possono usare meccanismi di reflection quali runtime type checking e downcasting, ma è preferibile definire per gli oggetti interessati un'opportuna superclasse o interfaccia comune e usare il polimorfismo. Per esempio per consentire di stampare i vincoli della `explanation` (`ArrayList<Constraint>`) di una soluzione ho fatto l'override del metodo `Object.toString` per le classi `EqConstraint` e `DiseqConstraint`.
Meccanismi di reflection sono necessari (e non basta il polimorfismo) quando si vuole fare runtime type checking bypassando il sistema dei tipi statico del linguaggio e quando si vuole analizzare la struttura di un oggetto a runtime. Per esempio i sistemi di ORM (object-relational mapping) come NHibernate per .NET fanno abbondante uso di reflection per analizzare la struttura degli oggetti da salvare su database.