UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI SCIENZE
MATEMATICHE, FISICHE e NATURALI
Corso di Laurea in Informatica

Tesi di Laurea

Implementazione di un'interfaccia standard per la programmazione a vincoli in Java basata sulla libreria JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Alberto Dallavalle

Anno Accademico 2009/2010

Ai miei genitori Rosalba e Silvano,
A mia sorella Stefania,
Ai miei amici.

Ringraziamenti

Un ringranziamento particolare va ai miei genitori e a mia sorella che mi hanno sempre sempre sostenuto e hanno atteso (ahimè così a lungo) questo mio traguardo.

Questo periodo universitario mi ha permesso di conoscere tante persone sempre disponibili ogniqualvolta ho avuto bisogno di una mano e con le quali confrontarsi nei momenti difficili. Ha formato un bel gruppo di amici che voglio qui ricordare e del quale vorrò sempre fare parte.

Ringrazio infine il mio relatore, Prof. Gianfranco Rossi, per la sua disponibilità, il suo aiuto e che ha permesso questo lavoro di tesi in tempi ristretti.

Indice

Capitolo 1

CSP e Constraint Programming

In questo capitolo verrà introdotta dapprima la nozione di Constraint Satisfaction Problem (CSP), per poi soffermarci maggiormente sulla risoluzione dei CSP e sulla integrazione della Constraint Programming (CP) nei linguaggi esistenti.

1.1 CSP

La nozione di Constraint Satisfaction Problem (CSP), o soddisfacimento di vincoli, nasce dalla ricerca in Intelligenza Artificiale (problemi di combinatoria, ricerca) e Computer Graphics.

Diversi modelli di pianificazione e di allocazione risorse, possono essere trattati come CSP e quindi risolti utilizzando questa tecnica.

Formalmente un CSP è definito da:

- un insieme di *variabili*, X_1,X_2,...X_n (n>0)
- un insieme di *vincoli*, C_1,C_2,...,C_m (m>0)

Ogni variabile X_i ha un *dominio* non vuoto D_i di possibili valori. Ogni vincolo C_i coinvolge un sottoinsieme delle variabili e ne specifica le combinazioni di valori consentite.

Uno *stato del problema* è definito dall'*assegnamento* di valori ad alcune o tutte le variabili, $X_i = v_i$, ..., $X_j = v_j$. Un assegnamento che non viola alcun vincolo è chiamato *consistente* o legale. Un assegnamento è *completo* se menziona tutte le variabili. Una *soluzione* di un CSP è un assegnamento completo che soddisfa tutti i vincoli. Alcuni CSP richiedono anche che la soluzione massimizzi una funzione obiettivo.

Può essere utile rappresentare un CSP come un grafo di vincoli, detto *constraint graph*, i cui nodi rappresentano le variabili(grandezze del problema) e gli archi i vincoli tra le variabili costituenti i nodi del grafo. Un vincolo unario che coinvolge una singola variabile X_i è rappresentato da un arco che inizia e termina sullo stesso nodo X_i definendo l'insieme di valori che la variabile può assumere. Un vincolo binario invece è rappresentato da un arco che collega due nodi X_i e X_h, definisce le coppie di valori che le variabili possono assumere contemporaneamente. Gli archi in un constraint graph possono essere orientati o meno a seconda della simmetria del vincolo. Ad esempio il vincolo $X_i \neq X_j$ può essere rappresentato da un arco non orientato, mentre il vincolo $X_i > X_j$ viene rappresentato da un arco orientato. In generale un vincolo può essere rappresentato come un arco non orientato se la relazione è simmetrica. La struttura del grafo può essere usata per semplificare il processo risolutivo, ottenendo in certi casi una riduzione esponenziale della complessità.

È facile vedere che si può dare una formulazione incrementale di un CSP come fosse un problema di ricerca standard.

- Stato iniziale: l'assegnamento vuoto , nel quale nessuna variabile ha un valore.
- Funzione successore: si può assegnare un valore a una qualsiasi delle variabili che non ce l'hanno ancora, a patto che non confligga con i valori assegnati in precedenza.
- Test obiettivo: l'assegnamento corrente è completo.
- Costo di cammino: un costo costante(per esempio 1) per ogni passo.

Infine ricordiamo una proprietà caratteristica fondamentale comune a tutti i CSP, la **commutatività**. Un problema è commutativo se l'ordine di applicazione di un qualsiasi insieme di azioni non ha effetto sul risultato finale. Questo è il caso dei CSP, dato che assegnando valori alle variabili si ottiene sempre lo stesso assegnamento parziale indipendentemente dall'ordine degli

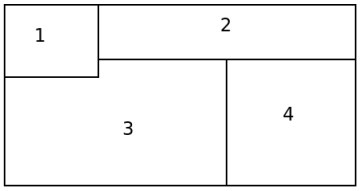

Figura 1.1: Esempio di mappa

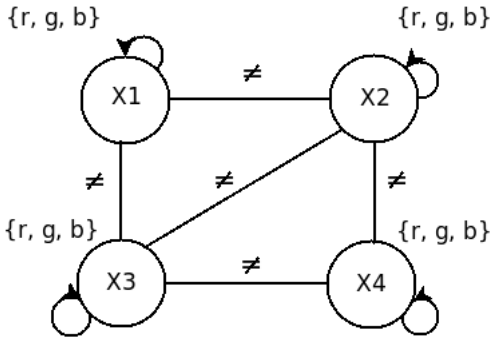

Figura 1.2: Constraing graph per l'esempio 1.1

assegnamenti. Di conseguenza, tutti gli algoritmi di ricerca CSP quando generano successori considerano i possibili assegnamenti di una sola variabile in ogni nodo dell'albero di ricerca.

Prendiamo un esempio: `Map-coloring`.

Supponiamo di dover colorare delle porzioni di un piano, denotate da un numero, in modo tale che due regioni contigue siano sempre colorate da colori diversi. Supponiamo anche di avere a disposizione i colori rosso(r), giallo(g) e blu(b). Il problema può essere rappresentato nel modo seguente: ogni regione `i` della mappa viene rappresentata da una variabile X_i che puo assumere i valori `r`, `g` e `b`. Dalla definizione del problem, due zone adiacenti non possono essere colorate con il medesimo colore, quindi le corrispondenti variabili sono legate dal vincolo diverso ($\neq$). In figura 1.1 è rappresentata una possibile mappa da colorare con quattro zone contrassegnate da un numero. Per colorare la mappa abbiamo a disposizione i tre colori suddetti. Il constraint graph corrispondente alla mappa di fugura 1.1 è rappresentato in figura 1.2.

1.2 Risoluzione di un CSP

Ogni soluzione è definita da un assegnamento completo alle variabili. Con n variabili il grafo dei vincoli avrà profondità n.

Un modo immediato per trovare un assegnamento consistente delle variabili ai valori contenuti nel loro dominio è quello di assegnare un valore di tentativo alle variabili e controllare, a posteriori, se il vincolo è soddisfatto. Se il controllo di consistenza fallisse, questo genererebbe così un backtracking e un'altra scelta di valori per le variabili. Questo metodo è detto *generate and test* in quanto prima genera un assegnamento casuale di valori per le variabili e solo successivamente controlla che i vincoli siano soddisfatti. Se l'insieme delle variabili del problema e l'insieme di vincoli sono grandi, il numero di tentativi da effettuare prima di trovare un assegnamento consistente con i vincoli è elevato e questo pregiudica in modo considerevole l'efficenza della ricerca della soluzione.

Esiste un metodo alternativo ed estremamente più efficiente che consiste nello sfruttare attivamente i vincoli (tramite la propagazione dei vincoli stessi) durante la fase di ricerca. Partendo da uno stato iniziale che contiene i vincoli dati nella descrizione del problema, il soddisfacimento di vincoli è un procedimento (eventualmente iterato) che consta di due passi. Il primo passo consiste nella propagazione dei vincoli. Al termine della propagazione o si è giunti ad una soluzione oppure è
necessario attuare un tentativo formulando, ad esempio, un'ipotesi sul valore di una variabile. Questo procedimento permette di ridurre considerevolemente il carico di ricerca da effettuare.

C'è da notare che spesso esiste più di un modo per modellare un problema. Per esempio, in un assegnamento dei posti a tavola per un incontro a cena, possiamo vedere gli invitati come variabili con ad ognuna associato lo stesso dominio, i posti a sedere. Ma avremmo potuto anche decidere di considerare i posti a sedere come variabili e la lista degli invitati il loro dominio. Le scelte effettuate in fase di modellazione del problema possono portare a una soluzione più o meno veloce dello stesso.

Nella maggior parte delle applicazioni pratiche, algoritmi CSP di uso generale possono risolvere problemi di ordine superiore a quelli trattati tramite algoritmi di ricerca generici.

In generale quindi in un algoritmo di soddisfacimento di vincoli, è necessaria la compresenza di due tipi di regole: quelle che definiscono come propagare correttamente i vincoli e quelle che suggeriscono quali tentativi effettuare per procedere quando non è più possibile propagare.

1.3 Vincoli FD

Nella forma più semplice di CSP vengono utilizzati vincoli a domini finiti, spesso chiamati più semplicemente *vincoli FD* in cui i domini delle variabili sono costituiti da intervalli di interi. I risolutori su questi vincoli risultano particolarmente efficienti nei problemi di schedulazione, ad esempio il Map Coloring e il problema delle n Regine. In questi problemi, quando la dimensione massima del dominio di ogni variabile del CSP è d, il numero possibile di assegnamenti è $O(d^n)$, ovvero esponenziale con il numero di variabili. Comunque nel caso pessimo non dobbiamo aspettarci di risolvere CSP a domini finiti in un tempo men che esponenziale.

1.4 Programmazione a vincoli

La **Constraint Programming (CP)** o programmazione a vincoli è un paradigma di programmazione che fornisce strumenti utili per trattare e risolvere al meglio problemi di soddisfacimento di vincoli. Fin dall'inizio si sono distinti due principali approcci:

1. estendere il linguaggio con appositi costrutti
2. aggiungere librerie che la supportino la CP.

Nel primo caso, sono stati definiti nuovi linguaggi di programmazione che estendono quelli esistenti. Ricordiamo DJ (Declarative Java) e Flow Java. Purtroppo, l'utilizzo di questi linguaggi è problematico, poichè sono di difficile integrazione con i sistemi esistenti e poco accettati dai programmatori.

L'alternativa è quella di aggiungere le funzionalità di CP ai linguaggi esistenti sottoforma di librerie, in particolare all'interno dei linguaggi Object-Oriented. Grazie all'incapsulamento e a meccanismi di overloading degli operatori, i programmatori possono così vedere i vincoli come reale parte del linguaggio. Alcune proposte in ambito Java sono: JSolver [6], Choco [7], JaCoP [8] , Koalog [9].

Queste diverse proposte utilizzano nomi diversi per rappresentare i concetti più importanti della CP, ma per risolvere un problema (CSP) seguono tutti questa struttura:

- Creazione del problema
- Creazione della variabili
- Aggiunta di vincoli sulle variabili
- Ricerca di una soluzione
- Ricerca di tutte le soluzioni

La Java Community così ha previsto una convenzione di denominazione unificata e una specifica dettagliata per questi concetti chiamata JSR-331 che verrà illustrata nel capitolo successivo.

Capitolo 2

JSR-331 Java Constraint Programming API

In questo capitolo verrà introdotta una specifica standard, denominata JSR-331, per la programmazione logica a vincoli nel linguaggio Java. Nella prima parte verranno descritti gli obiettivi prefissati per poi soffermarci sulla sua definizione.

JSR-331 (Java Specific Request) definisce una API Java per la programmazione a vincoli (CP) secondo quanto stabilito dalla Java Community. Questa specifica risponde alla necessità di ridurre i costi associati a incorporare la programmazione a vincoli nell'implementazione di applicazioni Java e di ottimizzare strumenti e servizi basati su questa tecnica.

2.1 Obiettivo

La standardizzazione della programmazione a vincoli mira a rendere questa tecnologia più accessibile per gli sviluppatori di software-business. Avere una interfaccia unificata permetterà lo sviluppo di applicazioni Java che potranno essere provate con diversi risolutori CP. Questo ridurrà la dipendenza da uno specifico fornitore favorendo uno sviluppo più semplice e libero.

Più in dettaglio gli obiettivi della specifica JSR-331 sono:

- Facilitare l'utilizzo della programmazione a vincoli nelle applicazioni Java.
- Aumentare la comunicazione e la standardizzazione tra i sviluppatori di CP.
- Incoraggiare la creazione di un mercato di strumenti e applicazioni basati su vincoli attraverso una CP API standard.

- Facilitare l'utilizzo della CP in altre specifiche JSR.
- Rendere le applicazioni Java più portabili da un risolutore di vincoli ad un altro.
- Fornire librerie di supporto ai vincoli per le applicazioni basate sulla CP.
- Supportare gli sviluppatori di CP, offrendo una API facilmente gestibile e che soddisfi le esigenze dei propri clienti.

2.2 Target Audience

Questa specifica è rivolta a tre maggiori utenze:

- gli sviluppatori di applicazioni che utilizzano la API standard o le interfacce Java per supporto alle decisioni.
- Fornitori di CP Solver che svilupperanno e manterranno in futuro le proprie implementazioni della API compatibili con la specifica.
- I ricercatori CP, che forniranno e miglioreranno le librerie dello standard, su vincoli, algoritmi di ricerca e su particolari problemi sostenuti dalla CP Community.

2.3 Campo d'utilizzo

La specifica è mirata alle piattaforme basate su Java ed è compatibile con jdk1.5 o superiore. Segue un approccio minimalista, ma dà particolare importanza alla facilità di utilizzo. Essa usa concetti già comunemente accettati e

le loro rappresentazioni sono state di fatto già standardizzate in vari solutori CP e articoli scientifici. Allo stesso tempo, la specifica è già sufficientemente ampia per consentire subito agli sviluppatori di applicazione di utilizzarla per risolvere tipici modelli CSP. Ci si aspetta che questa standardizzazione si estenda, aggiungendo sempre nuovi concetti CP, specifiche e strategie che verranno accettati dalla Community.

La JSR-331 si concentra solo sulla interfaccia CP e non assume alcun particolare approccio di implementazione. I seguenti concetti ed elementi sono al di fuori dell'ambito della specifica e restano una prerogativa dei diversi fornitori.

- meccanismi di implementazione sui diversi tipi di domini
- implementazione dei vincoli binari e globali
- meccanismo di propagazione dei vincoli
- meccanismi di backtracking.

2.4 Concepts

JSR-331 definisce tutti i concetti Java necessari per consentire a un utente di rappresentare e risolvere modelli CSP. JSR-331 supporta una netta demarcazione tra due parti diverse del CSP:

1. **Problem Definition** rappresentata dall'interfaccia Problem
2. **Problem Resolution** rappresentata dall'interfaccia Solver.

Tutti i concetti principali della CP appartengono ad una di queste due categorie. Ad alto livello tutto è rappresentato dalle seguenti interfacce:

- **Problem**
- Constrained Variable
- Constraint
- **Solver**
- Search Strategy
- Solution.

2.5 Interfaccia Problem

L'interfaccia `Problem` permette di rappresentare un CSP. Come tale è costituita da un insieme di variabili (istanze di `Var`) e un insieme di vincoli (istanze di `Constraint`). Ogni variabile ed ogni vincolo dipendono da un unico problema. Ad esempio il seguente codice:

```
Problem p = new Problem("Test");
Var x = p.variable("X",1,10);
```

crea un'istanza `p` della classe `Problem` (definita da una particolare implementazione JSR-331) e crea una nuova variabile intera `x` con il dominio [1,10] e nome X. Questa variabile è automaticamente aggiunta al problema. La JSR-331 utilizza l'interfaccia `Problem` come standard per definire i metodi principali che permettono all'utente finale di creare variabili e vincoli.

2.6 Interfaccia Var

Le variabili rappresentate da questa interfaccia sono le variabili di tipo intero. Ogni variabile di tipo `Var` ha un dominio finito di valori interi. Tutti i metodi che iniziano con la parola `variable` fanno si che le variabili siano sempre aggiunte al problema. Ad esempio:

```
Var v = p.variable("A",0,9);
```

Una nuova variabile con dominio $[0; 9]$ verrà creata e aggiunta al problema con il nome A. Questo significa che è sempre possibile trovare la variabile aggiunta usando il metodo `getVar(X)`. Un'alternativa per creare variabili intere senza aggiungerle al problema è utilizzare il costruttore definito nell'implementazione della classe `Var`. Ad esempio il codice visto in precedenza:

```
Var v = p.variable("A",0,9);
```

è equivalente al seguente codice: Var x = new Var(p,X,1,10); p.add(x); In qualsiasi implementazione dovrà essere presente il seguente costruttore della classe `Var`:

```
public Var(Problem p, String name, int min, int max)
```

L'interfaccia Var prevede i seguenti metodi per manipolare le variabili Var:

- int `getDomainSize()` ritorna il numero corrente di elementi del dominio

- boolean `isBound()` ritorna vero se la variabile è già istanziata con un singolo valore
- int `getValue()` ritorna il valore con il quale è stata istanziata
- int `getMin()` ritorna il minimo valore del dominio corrente
- int `getMax()` ritorna il massimo valore del dominio corrente

Sulle variabili `Var` è possibile definire vari vincoli. Ad esempio, se un utente vuole imporre il vincolo x+y¡10 può invocare il metodo `post` di `Problem`:

```
post(x.plus(y),"<",10);
```

il metodo `plus` implementa la somma tra variabili `Var` nella classe `Var`. Vediamo quali sono i metodi aritmetici definiti nell'interfaccia Var:

- `plus(int value)`: somma con un valore intero `value`
- `plus(Var var)`: somma con una variabile `var`
- `minus(int value)`: differenza con valore intero `value`
- `minus(Var var)`: differenza con una variabile `var`
- `multiply(int value)`: prodotto con un intero `value`
- `multiply(Var var)`: prodotto con una variabile `var`
- `divide(int value)`: divisione con un intero `value`
- `divide(Var var)`: divisione con una variabile `var`
- `mod(int value)`: somma con un intero `value`
- `sqr()`: prodotto della variabile con se stessa `var`
- `power(int value)`: variabile elevata al valore value `value`

Questi metodi creano solamente nuove variabili `Var` senza aggiungerle al problema. Se necessario possono essere aggiunte successivamente con il metodo `add(Var v)` della classe `Problem`.

All'utente può sembrare conveniente scrivere espressioni aritmetiche e creare vincoli su di esse, ma così facendo potrebbero crearsi troppe variabili o vincoli intermedi. Per rappresentare il vincolo $3x + 4y - 7x > 10$ un utente può rappresentarlo in questo modo:

```
Var exp = x.multiply(3).plus(y.multiply(4)).minus(z.multiply(7));
post(exp,">", 10);
```

Anche se è maggior efficiente scrivere così:

```
int [] coef1 = ( 3, 4, -7 );
Var[] vars = ( x, y, z );
post(coef1,vars,">", 10);
```

Vedremo maggiori dettagli sui vincoli e i metodi `post` nel paragrofo successivo.

2.7 Interfaccia Constraint

JSR-331 specifica molti metodi per trattare e definire relazioni tra le variabili. Questi metodi sono disponibili nell'interfaccia `Problem`. Alcuni esempi:

1. Un vincolo x<y tra due variabili si può esprimere come

   ```
   post(x,"<",y);
   ```

2. Per esprimere che la somma di tutte le variabili dell'array vars di tipo `Var[]` deve essere minore di 20 si scrive:

   ```
   post(vars,"<",20);
   ```

3. Per esprimere il seguente vincolo:

   ```
   3*x + 4*y -5*z + 2*t > x*y
   ```

   l'utente lo può creare in questo modo:

   ```
   Var xy = x.multiply(y); // non-linear
   int[] coefs = { 3, 4, -5, 2 };
   Var[] vars = { x, y, z, t };
   post(coefs, vars, ">", xy);
   ```

4. Per vincolare un insieme di variabili `Var vars` ad essere tutte diverse tra loro:

   ```
   postAllDifferent(vars);
   ```

Tutti i metodi della classe `Problem` che iniziano con la parola `post` creano e pubblicano i vincoli, essi vengono quindi aggiunti al problema. Al contrario per creare un nuovo vincolo senza aggiungerlo, utilizziamo il metodo `linear` anch'esso definito nella classe `Problem`. Ad esempio se scriviamo:

```
Constraint c1 = new Linear(x, ">", y);
```

Come vediamo viene creato il vincolo $x > y$ ma non viene aggiunto al problema corrente.

2.8 Interfaccia Solver

Per rappresentare la risoluzione di un problema, JSR-331 usa l'interfaccia Solver. Un Solver permette ad un utente di cercare la soluzione di un problema, quella ottimale, o tutte le soluzioni. Qui sotto vediamo un esempio:

```
problem.log("=== Find One solution:");
Solver solver = problem.getSolver();
Solution solution = solver.findSolution();
```

È possibile creare più Solver per lo stesso problema. Ognuno di loro può cercare soluzioni differenti. Viene creata infatti una nuovo oggetto `Solver` ogniqualvolta si chiama il metodo `getSolver` su una istanza `Problem`. In questo caso viene invocato il metodo il metodo `findSolution` di Solver che cerca, se esiste, la prima soluzione del problema. Se viene trovata una soluzione, il metodo `findSolution` crea un nuovo oggetto `Solution`.

2.9 Interfaccia Solution

Questa interfaccia standard specifica come le soluzioni possono essere create attraverso i metodi `find` della classe `Solver`. Vediamo un esempio d'utilizzo:

```
Solution solution = solver.findSolution();
if (solution != null)
solution.log();
else
problem.log("No Solutions");
```

Dopo che su una istanza `Solver` è stato invocato il metodo `findSolution`, se una soluzione viene trovata, viene ritornato un oggetto `Solution`. Su di esso viene invocato il metodo `log` che stampa le variabili ed i loro valori nella soluzione. Ad esempio di stampa è il seguente:

```
=== Find Solution:
Solution #1:
X[1] Y[4] Z[5] R[6]
```

Una istanza di questa classe contiene una copia di tutte le variabili che fanno parte della soluzione. È possibile quindi ricercare in una `solution` una variabile o il suo valore con i metodi `get` della classe `Solution`. Infine ad ogni soluzione rimane associato l'oggetto `Solver` con la quale è stata trovata.

Capitolo 3

JSetL

JSetL è una libreria Java che combina la programmazione Object-Oriented di Java con i concetti fondamentali della programmazione CLP, ed in particolare CLP($\mathcal{SET}$) come:

- variabili logiche,
- unificazione,
- strutture dati ricorsive,
- liste ed insiemi anche parzialmente specificati,
- unificazione,
- risoluzione di vincoli in particolare su insiemi,
- non-determinismo.

JSetL è stato sviluppato presso il Dipartimento di Matematica dell'Università di Parma. Si tratta di un package completamente scritto in codice Java. La libreria è un software libero, re-distribuibile e modificabile sotto i termini della licenza GNU. La versione corrente è JSetL 2.0.

Nell'implementazione sviluppata in questo lavoro di tesi si usano oggetti della libreria JSetL, che rappresentano le variabili a dominio finitovediamo quali:

- classe `IntLVar`
- classe `Constraint`
- classe `SolverClass`

3.1 IntLVar

Le varibili logiche intere `IntLVar` sono un particolare caso delle variabile logiche `Lvar`, nelle quali i valori si limitano ad essere dei numeri interi. In più una variabile logica intera può avere un dominio finito di valori e dei vincoli interi, aritmetici, associati ad essa.

Il dominio di una variabile logica intera può essere specificato quando una variabile `l` viene creata ed è automaticamente aggiornato quando vengono imposti e risolti vincoli su l in modo da mantenere la consistenza.

Ad esempio, se `x` e `y` sono due variabili logiche intere entrambe con dominio [1, 10] e viene aggiunto il vincolo x > y, il dominio di x viene aggiornato a
$[2, 10]$ e quello di y a $[1, 9]$. Quando il valore di una variabile è ristretto ad un singolo valore, alla variabile viene assegnato quel valore(`bound`). Invece se il dominio risulta vuoto, significa che il vincolo impostato non è soddisfacibile.

Il vincolo aritmetico imposto alla variabile logica è rappresentato da una congiunzione di vincoli atomici. Questi vincoli vengono generati valutando costruite da operatori di somma, sottrazione, prodotto, moltiplicazione e divisione applicati a variabili ed interi. Ad esempio se `e` è l'espressione aritmetica x.sum(y.sub(1)) dove `x` e `y` sono variabili logiche intere, la valutazione di `e` ritorna una nuova variabile logica intera X_1 con associato il vincolo X_1 = x + X_2 $\wedge$ X_2 = y - 1.

In JSetL, una variabile logica intera è un istanza della classe `IntLVar`, che estende la classe `LVar`. I valori delle variabile logiche sono interi. L'unione di intervalli, usati per rappresentare i domini delle variabili intere, sono istanze della classe `IntervalSet`.

Vediamo in dettaglio gli attributi di questa classe:

- **name**: contiene una stringa che corrisponde al nome esterno della `IntLvar`, se è stato specificato, altrimenti viene settato con la stringa `?`;
- **init**: flag che indica l'inizializzazione della variabile intera. Se è settato a `true` significa che la varibile logica è inizializzata, in caso contrario sarà `false`;
- **val**: contiene un eventuale valore della `IntLvar` nel caso in cui sia inizializzata oppure `null` in caso contrario.
- **domain**: settato inizialmente a `null` il quale conterrà il dominio della variabile. Tale dominio può essere definito nel momento della creazione con l'utilizzo dei costruttori definiti nella classe principale oppure quando i vincoli vengono risolti.
- **constraint**: contiene un eventuale vincolo aritmetico intero `Constraint` imposto alla variabile. Tale vincolo, viene costruito al momento della creazione con l'opportuno costruttore della classe `Constraint`.

Costruttori

IntLVar()
Crea una variabile logica intera con dominio [Integer.MIN VALUE,Integer.MAX VALUE]. Il vincolo associato a questa variabile è il vincolo vuoto.
IntLVar(String name)
Crea una variabile logica intera con nome `name` e dominio [Integer.MIN VALUE,Integer.MAX VALUE]. Il vincolo associato a questa variabile è il vincolo vuoto.
IntLVar(Integer i)
Crea una variabile logica intera inizializzata con il dominio[i,i]. Il vincolo associato è vuoto
IntLVar(String name, Integer i)
Crea una variabile logica intera con nome `name` e dominio [i,i]. Il vincolo associato è vuoto
IntLVar(IntLVar l)
Crea una variabile logica con dominio equivalente alla variabile logica `l`

IntLVar(String name, IntLVar l)
Crea una variabile logica con dominio equivalente alla variabile logica `l` e nome `name`
IntLVar(Integer a, Integer b))
Crea una variabile logica con dominio [a,b]. Se $b < a$ viene lanciata un'eccezione. Il vincolo associato è vuoto
IntLVar(String name, Integer a, Integer b)
Crea una variabile logica con dominio [a,b] e nome `name`. Se $b < a$ viene lanciata un'eccezione. Il vincolo associato è vuoto.

Gli oggetti `IntLVar` possono essere creati anche usando i metodi aritmetici `sum`, `sub`, `mul`, `div`, e `mod`.

Questi metodi sono invocati da oggetti `IntLVar` e ritornano oggetti `IntLVar`. Essi possono essere concatenati per formare espressioni.

IntLVar sum(Integer o)
Ritorna una variabile logica intera X_1 con associato un vincolo $X_1 = X_0 + \text{o} \wedge C_0$, dove X_0 è la variabile logica chiamante il metodo e C_0 il vincolo associato.

IntLVar sum(IntLVar v)
Ritorna una variabile logica intera X_1 con associato un vincolo $X_1 = X_0 + \text{v} \wedge C_v \wedge C_0$, dove X_0 è la variabile logica chiamante il metodo, C_0 il vincolo associato a X_0 e C_v il vincolo associato a `v`.

IntLVar sub(Integer o)
IntLVar sub(IntLVar v)
Come i metodi `sum` ma implementano la differenza.

IntLVar mul(Integer o)
IntLVar mul(IntLVar v)
Come i metodi `sum` ma implementano la moltiplicazione.

IntLVar div(Integer o)
IntLVar div(IntLVar v)
Come i metodi `sum` ma implementano la divisione.

IntLVar mod(Integer o)
IntLVar mod(IntLVar v)
Come i metodi `sum` ma implementano il resto della divisione intera.

La classe `IntLVar` prevede metodi che generano i vincoli aritmetici di comparazione. Vediamo quali:

Constraint eq(Integer o)
Ritorna un vincolo Constraint $X_0 = \text{o} \wedge C_0$ dove X_0 è la variabile logica chiamante e C_0 il vincolo associato.
Constraint eq(IntLVar v)
Ritorna il vincolo $X_0 = \text{v} \wedge C_0 \wedge C_v$, dove X_0 è la variabile chiamante, C_0 il vincolo associato e C_v il vincolo associato con la variabile logica `v`.

Constraint neq(Integer o)
Constraint neq(IntLVar v)
Come i metodi `eq` ma implementano $\neq$

Constraint le(Integer o)
Constraint le(IntLVar v)
Come i metodi `eq` ma implementano $\leq$

Constraint lt(Integer o)
Constraint lt(IntLVar v)
Come i metodi `eq` ma implementano $<$

Constraint ge(Integer o)
Constraint ge(IntLVar v)
Come i metodi `eq` ma implementano $\geq$

Constraint gt(Integer o)
Constraint gt(IntLVar v)
Come i metodi `eq` ma implementano $>$

3.2 Constraint

Nel paragrafo precedente abbiamo visto il funzionamento dell'oggetto `Constraint` in una variabile intera logica `IntLVar`. Vediamo più in dettaglio com'è definito.

In JSetL un vincolo è un oggetto della classe `Constraint` definito da questi 6 attributi principali che sono:

- **arg1, arg2, arg3, arg4**: campi di tipo `Object` in cui vengono memorizzati gli elementi coinvolti nel vincolo;
- **cons**: attributo di tipo intero in cui viene memorizzato il tipo di vincolo;
- **caseControl=0** attributo che viene utilizzato nel caso in cui la risoluzione del vincolo richieda una scelta non_deterministica.

Nel caso di vincoli binari vengono usati solo due degli attributi `Object` descritti. Un oggetto di tipo `Constraint` si distingue in due casi:

1. Constraint atomici
2. Constraint composti

Un `Constraint` atomico in JSetL è una espressione di questa forma:

- e_1.op(e_2);
- e_1.op(e_2,e_3);

dove `op` è un metodo predefinito (eq (=), neq ($\neq$), le ($\leq$), ge ($\geq$), lt ($<$), gt ($>$)) ed e_1, e_2, e_3 sono espressioni che formano con `op` un vincolo.

Invece un `Constraint` composito è definito come segue:

- c_1.and(c_2) ... .and(c_n)
- c_1.or(c_2) ... .or(c_n)

dove c_1,c_2...c_n sono `Constraint` atomici.

Il significato della scrittura c_1.and(c_2) ... .and(c_n) è la logica congiunzione $c_1 \wedge (c_2) \wedge ... \wedge (c_n)$, mentre c_1.or(c_2) ... .or(c_n) implementa l'OR logico $c_1 \vee (c_2) \vee ... \vee (c_n)$.

Ad esempio il codice seguente:

```
x.eq(y.sum(1)).and(x.eq(3)).and(y.neq(z))
```

implementa il `Constraint` $x = y + 1 \wedge x = 3 \wedge y \neq z$.

3.3 SolverClass

La classe `SolverClass` si occupa dell'introduzione dei vincoli nel constraint store e della loro risoluzione. È costituita da attributi protetti quindi accessibili solo dalla classe `SolverClass` stessa.
Il contraint store è realizzato mediante l'attributo `store` della classe `ConstraintStore` che estende la classe `Vector`, i cui elementi sono riferimenti a istanze della classe `Constraint`.

L'inserimento di un vincolo, nello store è realizzato tramite il metodo `add` della classe `SolverClass` a cui può viene passato un oggetto di tipo `Constraint`. Il metodo aggiunge il vincolo passato come parametro in coda allo `store`. Nel caso in cui il vincolo passato sia una congiunzione di vincoli, il metodo `add` aggiunge in coda al `ConstraintStore` tutti i vincoli della congiunzione.

Inseriti i vincoli nello `store`, questi vengono risolti invocando il metodo `solve` su una istanza della classe `SolverClass`.

Capitolo 4

Implementazione JSR-331 basata su JSetL

Nel capitolo seguente verrano illustrate le principali classi create per implementare l'interfaccia descritta nel documento di specifica JSR-331.

Come prima cosa è stato creato, nella cartella *src* di `JSetL`, un package denominato *cspApi*. Al suo interno sono state definite le classi principali per rispettare la specifica JSR-331. Si è cercato di creare delle classi che rispettino la struttura specificata, e per ognuna di esse, di sviluppare i metodi più importanti realizzabili con la libreria `JSetL`.

Le classi di JSR-331 implementate sono le seguenti:

1. Problem
2. Var
3. Constraint
4. Solver
5. Solution

Queste cinque classi verranno descritte in modo semplice e schematico nei paragrafi successivi. Inizialmente vengono descritti gli attributi di ogni classe, dopodichè, com'è stato possibile implementarle basandoci sulla libreria *JSetL* e alcuni esempi di utilizzo.

4.1 Classe Problem

Ricordiamo che un CSP o Constraint Satisfation Problem è un problema definito da:

- un insieme di variabili, X_1,X_2,...X_n, ognuna di dominio finito D_1,D_2,...D_n.
- un insieme di vincoli, C_1,C_2,...,C_m definiti su di esse.

Il soddisfacimento di questi vincoli porta alla soluzione del problema.
Per definire un CSP la specifica JSR-331 prevede l'utilizzo di una classe denominata `Problem`. Nel seguito presentiamo l'implementazione della classe `Problem` fatta utilizzando JSetL.

Campi dati

```
public class Problem {

    private String name = null;

    private Vector<Var> vars;

    private Vector<Constraint> constraints;

    private Solver solver=null;
}
```

dove:

- `String name`: se inizializzato dà il nome del problema, altrimenti è settato a `null`.
- `Vector <Var> vars`: campo `Vector` di `Var` che memorizza le variabili aggiunte al problema.
- `Vector<Constraint> constraints`: campo `Vector` di `Constraint` che memorizza i vincoli aggiunti al problema.
- `Solver solver`: campo di tipo `Solver`, viene istanziato solamente quando viene assegnato un risolutore al problema altrimenti è settato a `null`.

Come si può vedere, per rappresentare l'insieme delle variabili e l'insieme dei vincoli che caratterizzano un CSP, viene utilizzata la struttura dati `java.util.Vector`. Questa, ci permette di trattare l'aggiunta di nuove variabili e vincoli con facilità. L'oggetto `Solver`, viene utilizzato nella ricerca delle soluzioni.

Costruttori

È presente un solo costruttore public con un parametro che rappresenta il nome del problema. Il costruttore viene implementato nel seguente modo:

```
public Problem(String n) {
name = n;
vars = new Vector<Var>();
constraints = new Vector<Constraint>();
}
```

Per creare un nuovo problema è sufficiente quindi richiamare questo costrutture pubblico nel programma. Ad esempio:

```
Problem p = new Problem("Test");
```

crea una nuova istanza `p` della classe `Problem` con nome "`Test`".

Esiste un altro modo per istanziare la classe `Problem` senza dichiararla esplicitamente. Essa può essere ereditata dalla classe Java che rappresenta il programma finale da eseguire, in questo caso `Test`:

```
public class Test extends Problem {

    public static void main(String[] args) {
    Test p = new Test();

    ...

    }
}
```

Quando viene istanziato un nuovo oggetto `Test` come nell'esempio, viene creato di fatto un oggetto `Problem` con il costruttore di default seguente:

```
protected Problem(){
    name = "unknown";
    vars = new Vector<Var>();
    constraints = new Vector<Constraint>();
}
```

Così facendo quando si necessita l'oggetto `Problem` corrente, si utilizza l'oggetto implicito `this`.

Principali metodi public

Vediamo ora una veloce carrelata sui principali metodi public della classe `Problem` e l'implementazione di alcuni di essi utilizzando `JSetL`:

Metodi add

Tipo di ritorno	*Metodo*
Constraint	**add**(Constraint constraint) Aggiunge il vincolo constraint al problema e ritorna il vincolo.
Var	**add**(Var var) Aggiunge una variabile Var al problema e ritorna la variabile.

Nella loro implementazione, questi metodi aggiungono una variabile `Var` o un vincolo `Constraint` al problema semplicemente chiamando il metodo `add` sui rispettivi Vector della classe `Problem`:

```
public Constraint add(Constraint c){
    this.constraints.add(c);
    return c;
}
public Var add(Var v){
    this.vars.add(v);
    return v;
}
```

Metodi get

I metodi che iniziano con la parola `get` ritornano gli oggetti richiesti.

Tipo di ritorno	*Metodo*
Constraint	**getConstraint**(String name) Ritorna il vincolo con il nome name o null se non esiste nel problema.
String	**getName**() Ritorna il nome del problema.
Solver	**getSolver**() Ritorna l'istanza del Solver associata con questo problema.
Var	**getVar**(String name) Ritorna la variabile Var con il nome name o null se non esiste nel problema.
Var[]	**getVarArray**(String name) Ritorna l'array di variabili Var con il nome name o null se non esiste nel problema.
Var[]	**getVars**() Ritorna un array di variabili Var precedentemenenti aggiunte al problema.

Per quando riguarda la loro implementazione tramite `JSetL` vediamo in particolare il metodo `getSolver`:

```
public Solver getSolver(){
    if (solver == null) solver = new Solver(this);
    return solver;
}
```

Questo metodo assegna un `Solver` al problema se non e ancora stato assegnato precedentemente. Maggiori dettagli verranno ripresi nel paragrafo 4.4 di questo capitolo.

Metodi linear

I metodi linear costituiscono i costruttori di oggetti `Constraint` per la classe `Problem` senza che essi vengano pubblicati e quindi aggiunti al problema:

Tipo di ritorno	*Metodo*
Constraint	**linear**(Var var,String oper,int value) Crea il vincolo: var oper value senza aggiungerlo al problema.
Constraint	**linear**(Var var1,String oper,Var var2) Crea il vincolo: var1 oper var2 senza aggiungerlo al problema.

Vediamo la loro implementazione:

```
public Constraint linear(Var var,String oper,int value) throws Exception{
    return caseOper(var,oper,value);
}
public Constraint linear(Var var1,String oper,Var var2) throws Exception{
    return caseOper(var1,oper,var2);
}
```

Entrambi chiamano la funzione `caseOper` definita nella classe `Problem` che ritorna il nuovo oggetto `Constraint` richiesto. Per maggiori dettagli rimandiamo al paragrafo 4.3 di questo capitolo.

Metodi di stampa

Per la stampa a video sono stati implementati due metodi `log`:

Tipo di ritorno	*Metodo*
void	**log**(String text) Stampa la stringa text.
void	**log**(Var[] vars) Stampa le variabili contenute nell'array vars.

Guardiamo nel dettaglio l'implementazione del secondo metodo `log`:

```
public void log(Var[] array){
    System.out.print("Vars solution: ");
    System.out.println();
    System.out.print("[ ");
    System.out.println();
    if (this.vars.size() == 0)
    System.out.println("Empty");
    else {
    for (int i = 0; i < array.length; i++){
        array[i].getIntLVar().output();
        }
    }
    System.out.println(" ]");
    System.out.println();
}
```

Per stampare a video il valore o il dominio di una variabile `Var`, viene chiamata la funzione `output` sull'oggetto `IntLVar` contenuto in `Var`, estratto con il metodo protetto della classe `Problem` `getIntLVar()`.

Metodi di utilità su Var

La classe Problem offre i seguenti metodi per trovare il minimo o il massimo tra oggetti `Var`:

Tipo di ritorno	*Metodo*
Var	**max**(Var[] arrayOfVariables) Ritorna la variabile Var di valore massimo nell'array quando tutte sono istanziate.
Var	**max**(Var var1, Var var2) Ritorna la variabile Var di valore massimo tra le due quando entrambe istanziate.
Var	**min**(Var[] arrayOfVariables) Ritorna la variabile Var di valore minimo nell'array quando tutte sono istanziate.
Var	**min**(Var var1, Var var2) Ritorna la variabile Var di valore minimo tra le due quando entrambe istanziate.

Non diamo in dettaglio l'implementazione di questi metodi perche svolgono solo confronti tra
variabili Var istanziate ad un valore e la loro implementazione non risulta particolarmente interessante.

Vediamo invece dei metodi importanti per la creazione e la pubblicazione di vincoli ovvero oggetti di classe `Constraint`.

Metodi per la creazione di Constraint

Tipo di ritorno	*Metodo*
void	**post**(Constraint c) Pubblica il vincolo c.
Constraint	**post**(int[] array, Var[] vars,String oper,int value) Crea e pubblica il vincolo: array∗vars oper value.
Constraint	**post**(int[] array, Var[] vars,String oper,Var var) Crea e pubblica il vincolo: array∗vars oper var.
Constraint	**post**(Var[] vars, String oper, int value) Crea e pubblica il vincolo: somma delle variabili in vars oper value.
Constraint	**post**(Var[] vars, String oper, Var var) Crea e pubblica il vincolo: somma delle variabili in vars oper var.
Constraint	**post**(Var var, String oper, int value) Crea e pubblica il vincolo: var oper value.
Constraint	**post**(Var var1, String oper, Var var2) Crea e pubblica il vincolo: var1 oper var2.
Constraint	**post**(Var var1, Var var2, String oper, int value) Crea e pubblica il vincolo: (var1+var2) oper var2.
Constraint	**post**(Var var1, Var var2, String oper, Var var) Crea e pubblica il vincolo: (var1+var2) oper var.
Constraint	**postAllDifferent**(Var[] vars) Crea e pubblica un nuovo vincolo tale che tutte le variabili contenute in vars siano diverse tra loro.
Constraint	**postAnd**(Constraint c1, c2) Crea e pubblica il vincolo: c1 AND c2.
Constraint	**postMax**(Var[] vars, String oper, int value) Crea e posta il vincolo: max(vars) oper value dove max(vars) è il massimo dell'array vars.
Constraint	**postMax**(Var[] vars, String oper, Var var) Crea e posta il vincolo: max(vars) oper var dove max(vars) è il massimo dell'array vars.
Constraint	**postMin**(Var[] vars, String oper, int value) Crea e posta il vincolo: min(vars) oper value dove min(vars) è il minimo dell'array vars.
Constraint	**postMin**(Var[] vars, String oper, Var var) Crea e posta il vincolo: min(vars) oper var dove min(vars) è il minimo dell'array vars.
Constraint	**postOr**(Constraint c1, c2) Crea e pubblica il vincolo: c1 OR c2.

Molto utilizzato nei CSP è il vincolo generato dal metodo `postAllDifferent()` che impone la condizione che le variabiali contenute nell'array di `Var` passato come parametro siano tutte differenti. Vediamone l'implementazione utilizzando `JSetL`:

```
public void postAllDifferent(Var[] array) throws Exception {
    LSet set = LSet.empty().insAll(array);
    this.add(set.allDiff());
}
```

Per implementare questa funzionalità nell'implementazione, viene creata un'istanza di tipo `LSet` della libreria JSetL. In questo insieme inizialmente vuoto, vengono inseriti uno a uno gli elementi `Var` dell'array passato come parametro alla funzione `postAllDifferent`. Tutto ciò viene fatto per poter utilizzare il metodo `allDiff()` già presente nella libreria `JSetL` per gli oggetti di tipo `LSet`. Il metodo `allDiff()` ritorna un oggetto di tipo `JSetL.Constraint`. Per essere aggiunto al Vector dei vincoli del problema viene utilizzata un metodo particolare `add` protetto della classe **Problem** che trasforma l'oggetto di tipo `JSetL.Constraint` in un (`Constraint`) di JSR-331 e lo aggiunge al problema. Vediamo in dettaglio:

```
protected void add(JSetL.Constraint c){
    this.constraints.add(new Constraint(c,this));
    return;
}
```

Dove il nuovo oggetto `Constraint` è creato con il costruttore protetto di `Constraint`.

```
protected Constraint(JSetL.Constraint c,Problem p){
    this.constraints = new JSetL.Constraint();
    this.constraints.add(c);
    this.problem = p;
}
```

Ulteriori metodi `post` e qualche esempio verranno ripresi nel paragrafo 4.3 di questo capitolo.

Metodi vari

Per finire vediamo gli ultimi metodi della classe `Problem`:

Tipo di ritorno	*Metodo*
void	**setName**() Specifica il nome del problema.
void	**setSolver**() Richiama il metodo getSolver()
Var	**sum**(Var[] vars) Ritorna una nuova variabile Var, costruita dalla somma delle variabili nell'array vars.
Var	**variable**(int min, int max) Crea una nuova variabile Var con dominio [min,max].
Var	**variable**(String name,int min, int max) Crea una nuova variabile Var con nome name e dominio [min,max] e la aggiunge al problema.
Var[]	**variableArray**(String name,int min, int max, int size) Crea un nuovo array di variabili Var con nome name e dominio [min,max], lo aggiunge al problema e ritorna l'array.

I più significativi tra questi sono i metodi che iniziano con la parola `variable` che permettono la creazione di nuove variabili Var. Ne vedremo l'implementazione e l'uso nel paragrafo successivo.

4.2 Classe Var

La classe `Var` della specifica JSR-331 identifica una variabile intera a dominio finito. Questo tipo di variabile è implementata anche nella libreria JSetL con il nome di `IntLVar`.
L'implementazione della classe `Var` tramite JSetL si basa ovviamente sull'utilizzo di `IntLVar`. Inoltre in accordo con la specifica JSR-331 abbiamo ritenuto opportuno definire una classe `ConstrainedVariable` che serva da super-classe da cui derivare la classe `Var` e le eventuali future altre classi per altri di tipi di variabili.

Campi dati

```
public class ConstrainedVariable {

private String name = null;

private Problem p = null;

...
}

public class Var extends ConstrainedVariable{

private IntLVar ilv;

...
}
```

dove:

- `String name`: campo di tipo String che specifica il nome della variabile. Parametro ereditato dalla classe `ConstrainedVariable`.
- `IntLvar l`: campo di tipo IntLvar, rappresenta l'oggetto implementato tramite la libreria `JSetL`.
- `Problem p`: campo di tipo Problem, associa la variabile al problema in cui è definita. Parametro anch'esso ereditato dalla classe `ConstrainedVariable`.

Come vediamo, viene definito un oggetto IntLVar con la tecnica di incapsulamento. In questo modo nascondiamo il funzionamento dell'oggetto JSetL all'utente che utilizza oggetti di classe Var.

Creazione di Var

Nella classe `Var` è presente un costruttore con 4 parametri, Var(Problem p,String name,int a,int b) con cui si può costruire una nuova variabile intera con nome e dominio.

La sua implementazione tramite `JSetL` è la seguente:

```
public Var(Problem p,String name,int a,int b){
    ilv = new IntLVar(name,a,b);
    this.setName(name);
    this.setProblem(p);
}
```

Il costrutture public della classe `Var` deve inizializzare un oggetto di tipo `IntLVar` al suo interno quindi chiama a sua volta il costruttore della classe `IntLVar` di JSetL. Per completare l'istanza dell'oggetto `Var` viene associato infine il nome ed il problema passati come parametri al costruttore. Un esempio d'uso del costruttore è il seguente:

```
Var x = new Var(p,"X",1,10);
```

che crea un'istanza di tipo `Var` con dominio [1,10] e nome X. In questo modo viene creata una nuova variabile `Var` ma non viene pubblicata e quindi non viene aggiunta al problema.
Per farlo successivamente si utilizza il metodo `add` della classe `Problem` visto precedentemente:

```
p.add(x);
```

Come abbiamo visto precedentemente è anche possibile creare e pubblicare una nuova variabile utilizzando il metodo `variable` della classe `Problem`. Ad esempio:

```
Var x = p.variable("X",1,10);
```

Il problema crea un'istanza x di tipo Var con dominio[1,10] e nome X. Infine la aggiunge al suo insieme di variabili.
Vediamo il funzionamento nel dettaglio dell'implementazione tramite la libreria JSetL. Il metodo `variable` crea la variabile richiesta utilizzando il costruttore public della classe `Var`. Una volta creata la nuova `Var` viene aggiunta al problema con una semplice `add` della classe (`java.util.Vector`).

```
public Var variable(String s,int min,int max){
    Var v = new Var(this,s,min,max);
    vars.add(v);
    return v;
}
```

JSR-331 con il metodo `variableArray` della classe `Problem` permette di creare un array di variabili `Var` con dominio specificato. Ad esempio il codice seguente:

```
Var[] vars = variableArray("array",1,10,3);
```

Crea un nuovo array di 3 variabili `Var` con dominio [1,3].
Vediamo il funzionamento nel dettaglio dell'implementazione tramite la libreria JSetL.

```
public Var[] variableArray(String s,int min,int max,int size){
    Var[] v = new Var[size];
    for (int i=0;i<size;i++){
        String news = s + "[" + i + "]";
        v[i] = new Var(this,news,min,max);
        this.vars.add(v[i]);
}
return v;
}
```

In dettaglio viene costruito un array di tipo `Var` di dimensione `size` passata come parametro. Vengono create con il costruttore di `Var` nuove variabili con nome s[i] e dominio [min, max]. Con un `for` vengono aggiunte all'array e al problema. Infine il metodo ritorna l'array appena costruito.

Principali metodi public

Riassumiamo i metodi principali della classe Var implementati.

Metodi di utilità

Tipo di ritorno	*Metodo*
boolean	**contains**(int value) Ritorna true se il dominio della variabile contiene il valore value
boolean	**isBound**() Ritorna vero se questa variabile ha un solo valore.

Vediamo nel dettaglio la loro implementazione tramite `JSetL` :

```
public boolean contains(int value){
    return this.getIntLVar().getDom().contains(value);
}
public boolean isBound(){
    return this.getIntLVar().isKnown();
}
```

Entrambi i metodi utilizzano l'oggetto `IntLVar` al suo interno. Nel primo caso per verificare se il dominio della variabile contiene un certo valore vengono chiamati in sequenza i metodi della classe `IntLVar`, `getDom()` e `contains` rispettivamente per richiedere il dominio e verificare l'appartenenza del valore `value` passato come parametro.
Nel secondo caso, viene chiamato il metodo della classe `IntLVar isKnown()` per sapere se la variabile è istanziata ad un valore oppure no.

Metodi get

Tipo di ritorno	*Metodo*
int	**getDomainSize**() Ritorna il numero di valori contenuti nel dominio di questa variabile.
int	**getMax**() Ritorna il massimo valore del dominio di questa variabile.
int	**getMin**() Ritorna il minimo valore del dominio di questa variabile.
int	**getValue**() Ritorna il valore con il quale questa variabile è stata istanziata, altrimenti rilascia un'eccezione.

Anche in questo caso vediamo nel dettaglio la loro implementazione tramite `JSetL`:

```
public int getDomainSize() {
    return this.getIntLVar().getDom().size();
}
public int getMax(){
    return this.getIntLVar().getDom().getGlb();
}
```

```
public int getMin(){
    return this.getIntLVar().getDom().getLub();
}
public int getValue(){
    return this.getIntLVar().getValue();
}
```

Anche questi metodi agiscono sull'oggetto `IntLVar` di JSetL estratto con il metodo protetto `getIntLVar()` della classe `Var` per richiedere rispettivamente la dimensione, l'estremo superiore o inferiore di un dominio, o il valore istanziato.

Metodi aritmetici

Ora passiamo in rassegna alcuni metodi aritmetici:

Tipo di ritorno	*Metodo*
Var	**divide**(int value) Ritorna una nuova variabile, creata dalla divisione con il valore value.
Var	**divide**(Var var) Ritorna una nuova variabile, creata dalla divisione con la variabile var.
Var	**minus**(int value) Ritorna una nuova variabile, differenza tra la variabile chiamante e value.
Var	**minus**(Var var) Ritorna una nuova variabile, differenza tra la variabile chiamante e la variabile var.
Var	**mod**(int value) Ritorna una nuova variabile, divisione intera tra la variabile chiamante e l'intero value.
Var	**multiply**(int value) Ritorna una nuova variabile, prodotto tra la variabile chiamante e l'intero value.
Var	**multiply**(Var var) Ritorna una nuova variabile, prodotto tra la variabile chiamante e la variabile var.
Var	**plus**(int value) Ritorna una nuova variabile, somma tra la variabile chiamante e l'intero var.
Var	**plus**(Var var) Ritorna una nuova variabile, somma tra la variabile chiamante e la variabile var.
Var	**power**(int value) Ritorna una nuova variabile, prodotto tra la variabile chiamante e l'intero value.
Var	**sqr**() Ritorna una nuova variabile, prodotto tra la variabile chiamante per se stessa.

Tutti questi metodi sono implementati in egual modo ma vediamo in dettaglio l'implementazione del metodo `plus`.

```
public Var plus(Var exp){
    return new Var(this.getProblem(), this.getIntLVar().sum(exp.getIntLVar()));
}
```

Per sommare due variabili Var semplicemente viene chiamato il metodo `sum` tra i rispettivi oggetti `IntLVar`. Per ritornare un nuovo oggetto `Var` viene chiamato un costruttore protetto della classe `Var` così definito:

```
protected Var(Problem p,IntLVar l) {
    ilv = l;
    this.setProblem(p);
}
```

Altri esempi d'utilizzo verranno mostrati nei paragrafi successivi.

4.3 Classe Constraint

La classe `Constraint` della specifica JSR-331 identifica i vincoli definiti sulle variabili Var. Questo oggetti sono implementati anche nella libreria JSetL con il medesimo nome `Constraint`.
L'implementazione della nuova classe `Constraint` si basa sull'utilizzo degli oggetti `Constraint` della libreria `JSetL`.

Campi dati

```
public class Constraint {

    private String name = null;

    private JSetL.Constraint constraint;

    private Problem problem;
```

dove:

- `String name`: campo di tipo String, che specifica il nome del vincolo.
- `JSetL.Constraint constraint`: campo di tipo JSetL.Constraint, rappresenta gli oggetti Constraint implementati nella libreria JSetL.
- `Problem p`: campo di tipo Problem, associa il vincolo al problema in cui è definito.

A differenza dalle altre classi viste finora, non esistono costruttori public di questa classe. Vediamo quindi come sia possibile creare un nuovo vincolo.

Creazione oggetti Constraint

La creazione di oggetti `Constraint` avviene in due modi principali:

- creazione e pubblicazione di oggetti `Constraint` tramite metodi `post`.
- solo creazione di oggetti Constraint tramite metodi `linear`.

Metodi post

Come abbiamo visto precedentemente, tutti i metodi per creare e pubblicare vincoli fanno parte della classe `Problem` e iniziano con la parola `post`.
Vediamo alcuni esempi:

```
post(x,"lt",y); // uguale a postLinear(x,"lt",y);
post(x.plus(y),"=",z);
postAllDifferent(vars);
postElement(vars,indexVar, "eq", 5);
postCardinality(vars,3, "gt", 0);
```

Consideriamo in particolare il caso post(Var var1,String operator,Var var2).
Questo metodo può essere implementato tramite `JSetL` nel seguente modo:

```
public Constraint post(Var var1,String operator,Var var2) throws Exception{
    Constraint c = caseOper(var1,operator,var2);
    this.add(c);
    return c;
}
```

Per quanto riguarda il parametro `operator` della `post`, vengono riconosciuti i seguenti valori:

- **"eq"**: uguaglianza
- **"neq"**: diseguaglianza
- **"le"**: minore o uguale
- **"lt"**: minore stretto
- **"ge"**: maggiore o uguale

- **"gt"**: maggiore stretto

In base al valore di questa stringa, la funzione `caseOper` richiamata nel metodo `post` distingue l'operazione da settare nel nuovo vincolo. La stringa viene convertita nel codice operatore riconosciuto da uno `switch case` con il metodo `nametocode(String)` della classe `Enviroment` di `JSetL`.

```
protected Constraint caseOper(Var var1,String oper,Var var2) throws Exception{

    JSetL.Constraint cc = new JSetL.Constraint();


    switch(Environment.name_to_code(oper)){
        case(1): consistency(var1.eq(var2));
                 cc.and(var1.eq(var2));
        break;
        case(2): consistency(var1.neq(var2));
                 cc.and(var1.neq(var2));
        break;
        case(12):consistency(var1.ge(var2));
                 cc.and(var1.ge(var2));
        break;
        case(13):consistency(var1.gt(var2));
                 cc.and(var1.gt(var2));
        break;
        case(14):consistency(var1.le(var2));
                 cc.and(var1.le(var2));
        break;
        case(15):consistency(var1.lt(var2));
                 cc.and(var1.lt(var2));
        break;
    }
    return new Constraint(cc,this);
}
```

Per ogni caso possibile viene verificata la consistenza del vincolo costruito. Per fare ciò, è necessario far risolvere il vincolo ad un risolutore implicito. Questo compito è svolto dal metodo protetto `consistency` della classe `Problem`. In particolare viene istanziato un oggetto di tipo `SolverClass` della libreria `JSetL` e tramite la funzione `solve` si verifica la consistenza del vincolo. Nel caso in cui il vincolo non risulta soddisfacente viene rilasciata una eccezione.

```
protected void consistency(JSetL.Constraint cc) throws Exception{
    SolverClass s = new SolverClass();
    s.solve(cc);
    return;
}
```

Se il vincolo è consistente viene creato l'oggetto `Constraint` che lo rappresenta con il costruttore protetto:

```
protected Constraint(JSetL.Constraint c,Problem p){
    this.constraint = new JSetL.Constraint();
    this.constraint.and(c);
    this.problem = p;
}
```

La scrittura:

```
post(x.plus(y),"eq",z);
```

crea e pubblica il vincolo x+y=z. In questo caso, il parametro `var1` della `post` è ottenuto dalla valutazione di x.plus(y), che esprime la somma di due variabili `Var`, `x` ed `y`. Il metodo `plus` della classe `Problem` restituisce una nuova variabile `Var` ed è implementato nel modo seguente:

```
public Var plus(Var exp) {
     return new Var(this.getProblem(),this.getIntLVar().sum(exp.getIntLVar()));
    }
```

La funzione `plus` richiama il metodo `sum` di `JSetL` tra l'oggetto `IntLVar` del chiamante e l'oggetto `IntLVar` del parametro `exp` passato. La `sum` restituisce la somma
richiesta in una nuova variabile `IntLVar`. Quindi viene creato grazie ad un costruttore protetto della classe `Var`, un nuovo oggetto `Var`:

```
protected Var(Problem p,IntLVar l){
    this.ilv = l;
    this.setProblem(p);
}
```

Si noti che se una `post` non ha successo viene lanciata una eccezione. È regolare utilizzo postare vincoli all'interno di un blocco try-catch.

```
try {
post(x,"lt",y);
```

```
// X < Y
post(x.plus(y),"eq",z); // X + Y = Z
postAllDifferent(vars);
int[] coef1 = { 3, 4, -7, 2 };
post(coef1,vars,"gt",0); // 3x + 4y -7z + 2r > 0
} catch (Exception e) {
log("Error posting constraint: " + e);
System.exit(-1);
}
```

Metodi linear

A differenza di quanto visto finora, se volessimo costruire vincoli senza però aggiungerli al problema, la specifica JSR-331 prevede alcuni metodi `linear` nella classe `Problem`. Ad esempio:

```
Constraint c1 = linear(x,"lt",3);
Constraint c2 = linear(x.plus(y),"gt",z);
```

Consideriamo in particolare il caso linear(Var,String,Var). Questo metodo può essere implementato tramite `JSetL` nel seguente modo:

```
public Constraint linear(Var var1,String oper,Var var2) throws Exception{
    return caseOper(var1,operator,var2);
}
```

Come in precedenza, viene chiamata la funzione `caseOper` e di conseguenza il controllo di consistenza ma il vincolo non viene aggiunto al problema. Vediamo le function nominate: Se in un secondo momento volessimo aggiungere il vincolo basterà invocare una post dopo la linear:

```
Constraint c1 = linear(x,"lt",3);
post(c1);
```

Principali metodi public

Infine vediamo l'implementazione tramite `JSetL` dei metodi principali della classe `Constraint`:

Metodi get

Tipo di ritorno	*Metodo*
String	**getName**() Ritorna il nome del vincolo.
void	**setName**(*String name*) Assegna il nome name al vincolo chiamante.
Problem	**getProblem**() Ritorna il problema associato al vincolo.

L'implementazione di questi metodi prevede semplicemente l'accesso al corrispondente campo dati di `Constraint`.

Metodi per i connettivi logici

Tipo di ritorno	*Metodo*
Constraint	**and**(Constraint c) Ritorna una nuovo vincolo, AND tra il vincolo chiamante e c.
Constraint	**or**(Constraint c) Ritorna una nuovo vincolo, OR tra il vincolo chiamante e c.

Questi metodi realizzano l'AND o l'OR logico tra due oggetti `Constraint`. Queste funzionalità sono già presenti nella libreria JSetL. Vengono quindi chiamati i rispettivi metodi sugli oggetti JSetL.Constraint estratti con getConstraint(), metodo protetto della classe `Constraint`.

```
public Constraint and(Constraint c){
    return new Constraint (this.getConstraint().and(c.getConstraint()),
        this.getProblem());
}

public Constraint or(Constraint c){
    return new Constraint (this.getConstraint().or(c.getConstraint()),
        this.getProblem());
}
```

Metodo post

Tipo di ritorno	*Metodo*
void	**post**() Metodo utilizzato per postare il vincolo chiamante.

La funzione `post()` aggiunge un oggetto `Constraint` al problema corrente. Ad esempio:

```
Constraint c1 = linear(x,"lt",3);
c1.post();
```

Può essere implementata nel modo seguente:

```
public void post(){
   this.getProblem().add(this);
   return;
}
```

4.4 Classe Solver

La classe `Solver` della specifica JSR-331 gestisce la risoluzione di un problema. È possibile creare, tramite metodi specifici della classe, più solver per lo stesso problema. Questi solver possono produrre differenti soluzioni perseguendo obiettivi diversi.

L'implementazione creata sfrutta per la risoluzione un oggetto di tipo `SolverClass` della libreria JSetL.

Campi dati

```
public class Solver {

    private Vector<Solution> solutions;

    private SolverClass solver;

    private Problem problem;
```

dove:

- `Vector<Solution>solutions`: `Vector` di oggetti `Solution`, in cui vengono memorizzate le soluzioni trovate da questo specifico Solver.
- `SolverClass solver`: oggetto di tipo `SolverClass` della libreria JSetL.
- `Problem problem`: problema a cui il `Solver` è stato associato.

Creazione oggetti Solver

Nella specifica JSR-331 è possibile avere un'istanza del problema senza che nessun Solver sia creato. Infatti un'istanza di questa classe viene assegnata ad un problema solo quando viene chiamato il metodo `getSolver()` della classe `Problem` nel modo seguente:

```
Solver mysolver = getSolver();
```

Se non è ancora stata creata un'istanza `Solver` per il problema la `getSolver` provvede a crearne una tramite un costruttore protetto della classe `Solver`. Vediamo in dettaglio l'implementazione di questi metodi:

```
public Solver getSolver(){
    if (solver == null) solver = new Solver(this);
    return solver;
}
protected Solver(Problem p) {
    solutions = new Vector<Solution>();
    solver = new SolverClass();
    problem = p;
}
```

Principali metodi public

Vediamo ora l'implementazione dei principali metodi di `Solver`:

Tipo di ritorno	*Metodo*
Solution	**findSolution**() Ricerca e ritorna una soluzione del problema
Solution[]	**findAllSolutions**() Ricerca tutte le possibili soluzioni del problema

La `findSolution` viene implementata nel modo seguente:

```
public Solution findSolution(){

    if (solutions.size() == 0){

        addconditions();

    try{

        solver.solve();

    }catch (Exception e) {

        this.problem.log("=== No Solution");
        System.exit(-1);

        }

        return new Solution(problem.getVars(),this);
    }
    else if (solver.nextSolution())
     return new Solution(problem.getVars(),this);
    else return null;
}
```

Quando viene chiamata la funzione `findSolution()` su un oggetto di tipo `Solver` che non contiene già delle soluzioni, i vincoli del problema vengono aggiunti all'oggetto di tipo `SolverClass` della libreria JSetL contenuto al suo interno(campo `solver`). Queste operazioni sono realizzate dal metodo privato `addconditions()`:

```
private void addconditions(){

    Var[] vars = problem.getVars();
    Constraint[] constraints = problem.getConstraints();

    for(int i=0;i<constraints.length;i++){
        solver.add(constraints[i].getConstraint());
    }
    for(int i=0;i<vars.length;i++){
```

```
        solver.add(vars[i].getIntLVar().label());
}
```

Oltre ad aggiungengere i vincoli contenuti nel problema, per ogni variabile del problema viene aggiunto un `Constraint` di `labeling` creato con il metodo `label()`. Fare il labeling significa semplicemente assegnare ad una variabile un singolo valore del proprio dominio. Così facendo ogni variabile avrà uno valore fissato nella soluzione.

Infine viene chiamata la risoluzione all'oggetto `Solver` con il metodo `solve`. Se esiste una soluzione viene creato un nuovo oggetto `Solution` e viene aggiunto alle sue soluzioni. Per ogni variabile nel problema viene creata una nuova variabile `Var` con ugual nome ma con un definito valore del suo dominio e aggiunta al `Vector` di variabili rappresentante una soluzione.

4.5 Classe Solution

La classe `Solution` viene utilizzata per memorizzare le singole soluzioni al problema e stampare i risultati.

Campi dati

```
public class Solution {

private int number;

private Solver s;

private Vector<Var> varsolved;
```

dove:

- `int number`: campo intero che identifica il numero della soluzione.
- `Solver s`: campo di tipo Solver, identifica l'oggetto Solver con cui è stata trovata la soluzione.
- `Vector<Var>vsolved`: Vector di Var, che identifica le variabili e i rispettivi valori nella soluzione.

Creazione oggetti Solution

Come abbiamo già visto in precedenza, un'istanza di `Solution` viene creata solo chiamando un metodo risolutivo, come ad esempio `findSolution()` a una istanza `Solver`. Ad esempio:

```
Solver mysolver = getSolver();
Solution sol = mysolver.findSolution();
```

Prima viene associato un oggetto di tipo `Solver` al problema con il metodo `getSolver()` e poi viene chiesto di trovare una soluzione con il metodo `findSolution()`. La `findSolution` richiama al suo interno il costruttore di `Solution` così realizzato: Vediamo il dettaglio il costruttore per un nuovo oggetto `Solution`:

```
protected Solution(Var[] v,Solver s){
    this.varsolved = new Vector<Var>();
    for (int i=0;i<v.length;i++){
    varsolved.add(new Var(v[i].getIntLVar().getName(),
          v[i].getIntLVar().getValue(),s.getProblem()));
    }
    this.s = s;
    s.getSolutions().add(this);
}
```

Se una soluzione esiste, per stampare i risultati si chiama il metodo `log` sull'istanza di `Solution`:

```
sol.log();
```

Principali metodi public

Tipo di ritorno	*Metodo*
Var[]	**getVars**() Ritorna un array delle variabili presenti nella soluzione del problema
Var	**getVar**(String name) Ritorna la variabile della soluzione con il nome name
int	**getValue**(String name) Ritorna il valore nella soluzione della variabile con nome name
int	**getSolutionNumber**() Ritorna il numero associato alla soluzione
void	**log**() Stampa le variabili della soluzione

Tutte le variabili che formano una soluzione vengono memorizzate all'interno di un oggetto di tipo `Solution`. Quindi è possibile accedere agli oggetti `Var` semplicemente con un metodo `get` appropriato. Vediamo ad esempio l'implementazione di `getValue`:

```
public int getValue(String name){
    for (int i=0;i<varsolved.size();i++){
        if (varsolved.get(i).getName() == name) return varsolved.get(i).getValue
    }
}
```

Questo metodo restituisce il valore della variabile `Var` con nome `name` nella soluzione. Di seguito è riportata anche l'implementazione del metodo `log`:

```
public void log(){
    Var[] v = this.getVars();
    System.out.print("Solution:");
    System.out.print(number);
    System.out.println();
    for(int i=0;i<v.length;i++){
    v[i].getIntLVar().output();
    }
    System.out.println();
}
```

Nell'implementazione di questo metodo viene utilizzato il metodo `output()` della libreria JSetL sugli oggetti `IntLvar` contenuti.

4.6 Esempio d'utilizzo

Ora vediamo un semplice esempio completo scritto utilizzando le classi e i relativi metodi da noi implementati utilizzando `JSetL`:

```
public class Test extends Problem {

//DEFINIZIONE DELLE VARIABILI DEL PROBLEMA

public void define() {

    Var x = variable("X",1,10);

    Var y = variable("Y",1,10);

    Var z = variable("Z",1,10);

    Var r = variable("R",1,10);

    Var[] vars = { x, y, z, r };

//DEFINIZIONE DEI VINCOLI

    try {
        post(x,"lt",y); // X < Y

        post(z,"gt",4); // Z > 4

        post(x.plus(y),"eq",z); // X + Y = Z

        postAllDifferent(vars);

        int[] coef1 = { 3, 4, -5, 2 };

        post(coef1,vars,"gt",0); // 3x + 4y -5z + 2r > 0

        post(vars,"ge",15); // x + y + z + r >= 15
```

```
        int[] coef2 = { 2, -4, 5, -1 };

        post(coef2,vars,"gt",x.multiply(y)); // 2x-4y+5z-r > x*y

    } catch (Exception e) {
        log("Error posting constraints: " + e);
        System.exit(-1);
        }
}

// RISOLUZIONE DEL PROBLEMA

public void solve() {

    log("=== Find Solution:");

//ASSOCIA UN RISOLUTORE AL PROBLEMA

    Solver solver = getSolver();

//CERCA UNA SOLUZIONE
    Solution solution = solver.findSolution();

// STAMPA LA SOLUZIONE SE ESISTE
    if (solution != null)
        solution.log();
    else log("No Solution");

}

public static void main(String[] args) {

    Test p = new Test();
    p.define();
    p.solve();
}

}
```

In questo caso la classe `Test` viene derivata dalla classe `Problem`, quindi il problema non necessita di essere istanziato esplicitamente. Ogniqualvolta in un metodo si necessita dell'istanza `Problem` corrente verrà utilizzato l'oggetto implicito `this`.

Capitolo 5

Esempi

Vediamo ora due esempi completi che utilizzano l'implentazione JSR-331 per risolvere due classici problemi della programmazione a vincoli.

5.1 n Regine

Questo è forse il CSP più famoso. Viene chiesto di posizionare n regine su una scacchiera n×n (n≥3) in modo che le regine non siano sotto attacco tra di loro. Nessuna regina deve trovarsi quindi sulla stessa riga, colonna o diagonale con un'altra.

Il problema si può modellare con un CSP dove le n regine sono le variabili. Con x_i viene denotata la regina posizionata sulla colonna i-esima dove i = 1, 2, ..., n. Il dominio di ognuna delle variabili rappresenta l'insieme delle righe su cui può essere posizionata ciascuna regina quindi l'intervallo [1..n]. L'insieme dei vincoli può essere riassunto nel seguente modo: Per ogni i ∈ 1, 2, . . . , n - 1 j ∈ i + 1, . . . , n :

- $x_i \neq x_j$ (mai due regine sulla stessa riga)
- x_i - $x_j \neq$ i - j (mai due regine sulla stessa diagonale Sud-Ovest, Nord-Est)
- x_i - $x_j \neq$ j - i (mai due regine sulla stessa diagonale Nord-Ovest, Sud-Est).

Vediamo come può essere risolto il problema utilizzando con l'implementazione JSR-331 descritta nel capitolo precedente:

```
public class Queens extends Problem {

//Dimensione del Problema

public static final int N = 13;

    public static void main(String[] args) {
Queens q = new Queens();
q.define();
q.solve();
}



    public void define() {

//Dichiarazione e inizializzazione di N variabili
// con dominio finito[0,N-1].

Var[] queens = variableArray("Queen",0,N-1,N);

//======= Definizione e post dei vincoli

try {

//Chiamata della funziona AllDifferent per gli N-1 oggetti Var creati

 postAllDifferent(queens);

//Post dei vincoli necessari per la risoluzione dell'algoritmo

 for(int i = 0; i < N-1; i++)
       for(int j = i+1; j < N; j++){
       post(queens[j].plus(j).minus(queens[i]),"neq",i);
       post(queens[i].plus(j).minus(queens[j]),"neq",i);
      }

} catch (Exception e) {
log("Error posting constraints: " + e);
System.exit(-1);
}
```

```
}

// Risoluzione del problema

public void solve() {
log("=== Find Solution:");

// Assegnazione di un Solver al problema e ricerca della soluzione

Solver solver = getSolver();
Solution solution = solver.findSolution();
solution.log();
}

}
```

Questo codice porterà al seguente risultato:
=== Find Solution:
Solution:1
Queens[0] = 0 Queens[1] = 2 Queens[2] = 4 Queens[3] = 1 Queens[4] = 8 Queens[5] = 11 Queens[6] = 9 Queens[7] = 12 Queens[8] = 3 Queens[9] = 5 Queens[10] = 7 Queens[11] = 10 Queens[12] = 6

5.2 Map Coloring

Il Map Coloring è un problema ben noto in letteratura che consiste nello stabilire se sia possibile colorare una mappa geografica utilizzando solo un numero definito e limitato di colori dove due paesi confinanti non possono avere lo stesso colore.

Molti problemi di allocazione di risorse possono essere ricondotti a questo problema. Il problema è stabilire quando i nodi possono essere colorati utilizzando solo k colori in modo che ogni coppia di nodi adiacente non abbia lo stesso colore.

Una possibile soluzione del problema utilizzando l'implementazione JSR-331 mostrata in precedenza è la seguente:

```
public class MapColoring extends Problem {

static final String[] colors = { "red", "green", "blue" };
```

```
public static void main(String[] args){
        MapColoring c = new MapColoring();
        c.solve();
    }


public void solve() {

// Inizializzazione Variabili

int n = colors.length-1;
Var Belgium = variable("Belgium",0, n);
Var Denmark = variable("Denmark",0, n);
Var France = variable("France",0, n);
Var Germany = variable("Germany",0, n);
Var Netherlands = variable("Netherland",0, n);
Var Luxemburg = variable("Luxemburg",0, n);
Var[] vars = {Belgium,Denmark,France,Germany,Netherlands,Luxemburg};

//======= Definizione e post dei vincoli
try {

post(France,"neq",Belgium);
post(France,"neq",Germany);
post(Belgium,"neq",Netherlands);
post(Belgium,"neq",Germany);
post(Germany,"neq",Netherlands);
post(Germany,"neq",Denmark);

} catch (Exception e){
log("Error posting constraints: " + e);
System.exit(-1);
}

Solution solution = getSolver().findSolution();
if (solution != null) {

// Stampa la soluzione trovata
solution.log();
```

```
// Associa ai valori trovati i colori corrispondenti
for (int i = 0; i < vars.length; i++) {
String name = vars[i].getName();
log(name+" - "+colors[solution.getValue(name)]);
}
}
else log("no solution found");

}

}
```

Viene visualizzato il seguente risultato:
Solution:1
`Belgium` = 0 `Denmark` = 0 `France` = 1 `Germany` = 2 `Netherland` = 1 `Luxemburg` = 0
`Belgium` - red `Denmark` - red `France` - green `Germany` - blue `Netherland` - green `Luxemburg` - red

Capitolo 6

Conclusioni e lavori futuri

In questo lavoro abbiamo affrontato il problema dell'implementazione di una API che permetta la risoluzione di modelli CSP secondo la specifica JSR-331 utilizzando gli strumenti per la programmazione dichiarativa a vincoli offerti dalla libreria `JSetL`.

Il risultato ottenuto è un nuovo pacchetto denominato cspApi che offre un supporto alla programmazione a vincoli nel rispetto dello standard definito dalla specifica JSR-331.

L'implementazione è risultata relativamente semplice ed efficiente dal momento che molte delle funzionalità previste in JSR-331 hanno un corrispondente immediato nella libreria `JSetL`. L'integrazione con `JSetL` è stata fatta in modo da essere del tutto trasparente all'utente finale.

Alcune possibilità previste nella specifica JSR-331, come ad esempio vincoli su variabili reali e strategie più sofisticate di ricerca di una soluzione di un problema CSP, non sono state prese in considerazione perchè richiederebbero un'estensione non banale alle funzionalità offerte da `JSetL` e potranno essere oggetto di futuro lavoro.

Bibliografia

[1] Stuart Russel, Peter Norvig
Intelligenza Artificale, un approccio moderno
Pearson, 2005

[2] Luca Console, Evelina Lamma, Paola Mello, Michela Milano
PROGRAMMAZIONE LOGICA E PROLOG
Utet, 1997

[3] Jacob Feldman
JSR-331 Java Constraint Programming API SPECIFICATION

[4] G.Rossi, E.Panegai, E.Poleo
JSetL: a Java library for supporting declarative programming in Java

[5] Krzysztof R. Apt
Principles of Constraint Programming
CWI, Amsterdam, The Netherlands

[6] *JSOLVER*
http://comp.rgu.ac.uk/staff/rab/CM3002/Java/JSolver/JSolver2.0Intro.pdf/

[7] *CHOCO*
http://choco.sourceforge.net/

[8] *JaCoP*
http://www.jacop.eu/

[9] *Koalog*
http://freshmeat.net/projects/koalogjcs/