UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE

MATEMATICHE, FISICHE e NATURALI

Corso di Laurea in Informatica

Tesi di Laurea Triennale

Progettazione e realizzazione in Java di un risolutore di vincoli su domini finiti

Candidato:

Daniele Pandini

Relatore:

Prof. Gianfranco Rossi

Correlatore:

Dott. Federico Bergenti

Anno Accademico 2006/2007

Ai miei genitori,
ai miei nonni,
ai miei fratelli
a Giulio
e agli altri miei amici

Indice

Capitolo 1

Introduzione

I problemi di soddisfacimento di vincoli (si veda ad esempio [4]) sono caratterizzati in generale da due importanti componenti. La prima è che tutti i problemi a vincoli devono includere **variabili**: oggetti che possono assumere diversi valori. L'insieme di questi possibili valori per ciascuna variabile è chiamato il **dominio** della variabile. Per esempio, in un problema di 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.

La seconda componente di ogni problema a vincoli è l'**insieme dei vincoli**. I vincoli sono regole che impongono limitazioni sui valori delle variabili. Con riferimento all'esempio precedente, se il padrone e la padrona di casa debbono sedere ai due capotavola, la loro scelta di posto è vincolata.

Nel corso degli anni sono stati studiati e realizzati vari **risolutori di vincoli**, alcuni a se stanti, altri "ospitati" in linguaggi logici o in linguaggi *object-oriented*.

Un approccio molto utilizzato per la risoluzione di vincoli è quello dei cosiddetti **Vincoli a Domini Finiti**, spesso chiamati più semplicemente vincoli $\mathcal{FD}$ [1, 3], in cui i domini delle variabili sono costituiti da intervalli

di interi ed i vincoli sono le normali relazioni di confronto su espressioni aritmetiche. Tali risolutori risultano essere particolarmente efficienti su molti problemi di soddisfacimento di vincoli, come ad esempio il *map coloring* e, più in generale, *problemi di schedulazione*. Uno dei sistemi più noti basato basato sugli $\mathcal{FD}$ è CHIP [2], un linguaggio logico a vincoli sviluppato a metà degli anni ottanta.

Un altro tipo di approccio alla programmazione con vincoli che ha avuto di recente un discreto successo è quello costituito dai cosiddetti **Vincoli Insiemistici**, in cui i domini delle variabili sono costituiti da insiemi finiti ed i vincoli sono le normali operazioni insiemistiche. Una proposta che si inserisce in questo filone di ricerca è quella di JSetL [11, 13], una libreria Java per il un supporto alla programmazione dichiarativa sviluppata all'interno del Dipartimento di Matematica dell'Università di Parma, che offre tra l'altro un risolutore di vincoli su insiemi del tutto simile a quello sviluppato per il linguaggio CLP(SET) [8].

Il maggior problema che si ha con la tecnica di risoluzione dei vincoli usata in JSetL è che l'albero di ricerca della soluzione generato è spesso troppo ampio per poter essere esplorato in tempo ragionevole. Infatti in JSetL non viene applicata nessuna riduzione dei domini delle variabili, come quelle tipicamente usate con i vincoli FD, ma viene utilizzata una strategia di ricerca della soluzione di tipo *generate & test*. Con questa tecnica viene prima generata una ipotetica soluzione assegnando ad ogni variabile uno dei possibili valori che ha nel proprio dominio e poi viene controllata la consistenza di ogni vincolo. Se poi ogni vincolo risulta consistente si eseguono alcuni test di consistenza globale, altrimenti si ha un fallimento e si procede con la generazione di un'altra ipotetica soluzione.

L'obbiettivo che ci prefiggiamo con questo lavoro di tesi è quello di creare una nuova libreria Java, detta JSetL($\mathcal{FD}$), in grado di integrare un nuovo risolutore di vincoli su domini finiti con il risolutore JSetL già esistente. Tale

integrazione dovrebbe permettere di mantenere i vantaggi di generalità e di potere espressivo dei vincoli di JSetL, e, nel contempo, permettere un trattamento efficiente dei vincoli aritmetici come quello fornito nei risolutori FD. Più precisamente, JSetL($\mathcal{FD}$) dovrebbe offrire i seguenti vantaggi rispetto a JSetL:

- Possibilità di trattare **vincoli aritmetici** e disuguaglianze su interi come veri vincoli, risolvibili anche quando parte degli operandi non hanno un valore noto.
- Maggiore **efficienza di risoluzione** di problemi che possono essere modellati come problemi a vincoli su domini finiti e che contengono vincoli aritmetici. La riduzione dei domini delle variabili fornita dalle tecniche $\mathcal{FD}$ riduce lo spazio di ricerca della soluzione del problema stesso.
- Maggiore **efficienza nella verifica della consistenza**. Le informazioni sui domini delle variabili facilitano la propagazione dei vincoli.
- Maggiore efficacia e completezza nella verifica di vincoli di **cardinalità insiemistica**. Infatti per tali vincoli è richiesta la capacità di trattare sistemi relativamente complessi di equazioni e disequazioni su interi.

La dissertazione è organizzata come segue.

Capitolo 2: Programmazione a vincoli: Nozioni ed Esempi. Questo capitolo è diviso in due parti: nella prima parte viene presentata un'introduzione informale alla programmazione a vincoli e vengono mostrati alcuni esempi comuni di problemi che possono essere modellati come **problemi di soddisfacimento di vincoli** detti più brevemente **CSP** (dall'Inglese: *constraints satisfaction problems*); nella seconda parte vengono introdotti i concetti matematici utilizzati nel resto del lavoro.

Capitolo 3: Risolutore di Problemi a Vincoli su Domini Finiti. In questo capitolo è descritta la teoria su cui si basa la progettazione e l'implementazione del risolutore di problemi a vincoli su domini finiti JSetL($\mathcal{FD}$).

Capitolo 4: JSetL($\mathcal{FD}$). In questo capitolo viene prima data una breve introduzione alla libreria JSetL e poi vengono descritte quali sono stati i vantaggi del passaggio a JSetL($\mathcal{FD}$).

Capitolo 5: Realizzazione di JSetL($\mathcal{FD}$). In questo capitolo vengono presentate le principali scelte di progettazione per la realizzazione del risolutore a vincoli su domini finiti descritto nel capitolo precedente e per la sua integrazione con il risolutore su vincoli insiemistici, presente in JSetL.

Capiolo 6: Un'estensione a JSetL($\mathcal{FD}$): il vincolo `allDifferent`. Nella prima parte di questo capitolo viene fatta un'introduzione al vincolo `allDifferent` e al suo utilizzo nella programmazione a vincoli. Nel resto del capitolo viene descritta l'implementazione di `allDifferent` in JSetL($\mathcal{FD}$).

Capitolo 7: Esempi, Test e Confronti. Questo capitolo è suddiviso in due parti. Nelle prima sono mostrati due esempi di programmi completi che utilizzano la libreria JSetL($\mathcal{FD}$) per risolvere due classici problemi della programmazione a vincoli. Nella seconda vengono posti a confronto i tempi di risposta per il problema delle n regine scritto con JSetL($\mathcal{FD}$) e con SWI Prolog [19]

Capitolo 8: Conclusioni. Questo capitolo contiene riassunto il lavoro da noi svolto in questa tesi e un accenno ai lavori futuri.

Capitolo 2

Programmazione a vincoli: Nozioni ed Esempi

Questo capitolo è diviso in due parti: nella prima parte verrà presentata un'introduzione informale alla programmazione a vincoli e verranno mostrati alcuni esempi comuni di problemi che possono essere modellati come **problemi di soddisfacimento di vincoli** detti più brevemente **CSP** (dall'Inglese: *constraints satisfaction problems*); nella seconda parte verranno introdotti i concetti matematici utilizzati nel resto del testo. Per maggiori informazioni si veda [4, 5].

2.1 Primi Concetti ed Esempi

In generale, i problemi di soddisfacimento di vincoli sono caratterizzati da due importanti componenti. Primo, tutti i problemi a vincoli devono includere **variabili**: oggetti che possono assumere diversi valori. L'insieme di questi possibili valori è chiamato il **dominio** della variabile. Per esempio, nell'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.

La seconda componente di ogni problema a vincoli è l'**insieme dei vincoli**. I vincoli sono regole che impongono limitazioni sui valori delle variabili. Se il padrone e la padrona di casa debbono sedere ai due capotavola, la loro scelta di posto è vincolata. Se ogni coppia deve sedere vicina o uno di fronte all'altra, allora bisogna includere il vincolo nella formulazione del problema. Non dimentichiamo poi che due invitati non possono cenare seduti in due su di una sedia e di aggiungere questo vincolo al problema.

Notare che spesso c'è più di un modo per **modellare un problema**. Nel precedente esempio, avremmo potuto 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. Di seguito vediamo alcuni esempi di problemi a vincoli tra i più classici. Ulteriori esempi possono essere trovati nel sito Web CLPLib [7].

2.1.1 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. Questo problema è conosciuto nel campo della teoria dei grafi come *k-colorability*. Molti problemi di allocazione di risorse possono essere ricondotti a questo problema. Un esempio di questo è il problema dell'assegnamento delle frequenze radio, un problema di telecomunicazione dove l'obiettivo è quello di assegnare frequenze radio alle stazioni in modo che tutte possano operare contemporaneamente senza udibili interferenze.

Utilizzando la teoria dei grafi, la mappa geografica può essere astratta come un grafo con un nodo per ogni paese e un arco congiungente ogni coppia di paesi confinanti. Dato un grafo di grandezza arbitraria, 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.

Il problema di *k-colorability* dei grafi è formulato come un CSP dove ogni nodo nel grafo è una variabile e il dominio associato alle variabili sono i possibili colori, mentre l'insieme delle condizioni imposte sulle coppie di vertici adiacenti è l'insieme dei vincoli.

2.1.2 Problemi di Schedulazione

La schedulazione è un problema che si presta naturalmente ad essere modellato con i CSP. I problemi di schedulazione consistono nell'assegnamento di attività o compiti quando le risorse sono limitiate e si hanno vincoli di precedenza temporale. Ad esempio la schedulazione delle classi e degli insegnanti in una università. Solitamente in questo tipo di problemi vengono associati i compiti con le variabili e i loro domini sono l'istante temporale di inizio del compito.

2.1.3 Problemi Cripto Aritmetici ($SEND + MORE = MONEY$)

I problemi cripto aritmetici sono puzzle matematici in cui le cifre sono sostituita da lettere dell'alfabeto o da simboli. Un classico esempio spesso citato in ambito di programmazione a vincoli è il problema:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

dove le lettere devono essere rimpiazzate con cifre da 0 a 9 in modo che l'equazione sia corretta. Qui le variabili del problema sono le lettere S, E, N, D, M, O, R, Y. Poiché S e M compaiono come ultime cifre a sinistra, non possono essere 0 e quindi hanno come dominio l'intervallo di numeri interi

da 1 a 9 che esprimiamo come [1..9]. Le variabili rimanenti hanno invece dominio [0..9]. I vincoli del problema danno origine all'equazione aritmetica

$$\begin{aligned} & 1000 \cdot S + 100 \cdot E + 10 \cdot N + D \\ + \; & 1000 \cdot M + 100 \cdot O + 10 \cdot R + E \\ = 10000 \cdot M + \; & 1000 \cdot O + 100 \cdot N + 10 \cdot E + Y \end{aligned}$$

e impongono che ogni variabile sia diversa dalle altre. Un altro modo di modellare il problema è tramite l'introduzione di variabili 'resto' con dominio associato l'intervallo [0..1] ed utilizzando al posto di un'unica equazione aritmetica, le cinque equazioni seguenti

$$\begin{aligned} D + E &= 10 \cdot C_1 + Y, \\ C_1 + N + R &= 10 \cdot C_2 + E, \\ C_2 + E + O &= 10 \cdot C_3 + N, \\ C_3 + S + M &= 10 \cdot C_4 + O, \\ C_4 &= M. \end{aligned}$$

C_1, C_2, C_3, C_4 sono le variabili resto. Entrambi i CSP hanno un unica soluzione:

$$\begin{array}{rr} & 9567 \\ + & 1085 \\ \hline & 10652 \end{array}$$

2.1.4 Problema delle n Regine

Questo è forse il CSP più famoso. Viene chiesto di posizionare n regine su una scacchiera $n \times n$, $(n \geq 3)$, in modo che le regine non siano sotto scacco 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 $i = 1, 2, \ldots, n$. Il dominio di ognuna delle variabili rappresenta l'insieme delle

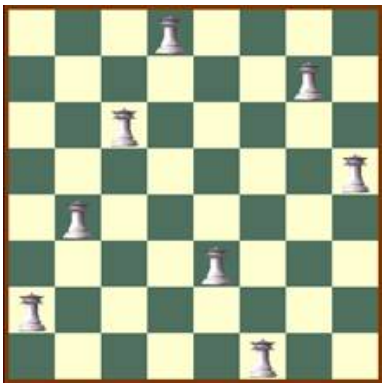


Figura 2.1: Una delle 92 soluzioni per il problema delle 8 regine

righe su cui può essere posizionata ciascuna variabile e quindi l'intervallo $[1..n]$. L'insieme dei vincoli può essere riassunto nel modo seguente.

Per ogni $i \in 1, 2, \ldots, n-1,\ j \in i+1, \ldots, n$:

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

2.2 Concetti matematici preliminari

2.2.1 Nozioni di base

Sia x una variabile, il **dominio** di x è denotato da D_x ed è l'insieme di valori che possono esse assegnati a x. Come semplificazione della definizione $D_x = \{d_1, d_2, \ldots, d_n\}$ scriveremo spesso $x \in \{d_1, d_2, \ldots, d_n\}$. In questo scritto considereremo solo variabili con **domini finiti** detti più brevemente $\mathcal{FD}$.

Sia $\mathcal{Y} = y_1, y_2, \ldots, y_k$ una sequenza finita di variabili dove $k > 0$. Un **vincolo** C su $\mathcal{Y}$ è definito come il sottoinsieme del prodotto Cartesiano dei domini delle variabili in $\mathcal{Y}$, ad esempio $C \subseteq D_{y_1} \times D_{y_2} \times \cdots \times D_{y_k}$. Questo è scritto come $C(\mathcal{Y})$ oppure $C(y_1, y_2, \ldots, y_k)$. Quando $k = 1$ diremo che il vincolo è un **vincolo unario**, quando $k = 2$ diremo che è **un vincolo binario**, mentre con **vincolo globale** indicheremo vincoli definiti su più di due variabili.

Un **problema di soddisfacimento di vincoli (CSP)** è definito da una sequenza di variabili $\mathcal{X} = x_1, x_2, \ldots, x_n$ con rispettivi domini $\mathcal{D} = D_{x_1}, D_{x_2}, \cdots D_{x_n}$ e un insieme finito di vincoli $\mathcal{C}$ in cui ogni vincolo è definito su un sottoinsieme di $\mathcal{X}$. Un CSP P è denotato da $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$.

Sia $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un CSP dove $\mathcal{X} = x_1, x_2, \ldots, x_n$ e $\mathcal{D} = D_{x_1}, D_{x_2}, \cdots D_{x_n}$. Una tupla $(d_1, d_2, \ldots, d_n) \in D_{x_1} \times D_{x_2} \times \cdots \times D_{x_n}$ *soddisfa* un vincolo $C \in \mathcal{C}$ sulle variabili $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ se $(d_{i_1}, d_{i_2}, \ldots, d_{i_m}) \in C$. Se nessuna tupla soddisfa C, diremo che C è *inconsistente*. Una tupla $(d_1, d_2, \ldots, d_n) \in D_{x_1} \times D_{x_2} \times \cdots \times D_{x_n}$ è una *soluzione* di CSP se soddisfa tutti i vincoli $C \in \mathcal{C}$.

Un CSP è detto **consistente** se è un CSP per cui esiste una soluzione. Un CSP è detto **inconsistente** se è un CSP per cui non esiste una soluzione. Un CSP si dice **fallito** se è un CSP in cui una o più delle sue variabili hanno dominio vuoto oppure dove le sue variabili hanno solo domini singoletti che non sono soluzione del CSP. Un CSP è detto **risolto** se è un CSP dove le sue variabili hanno tutte domini singoletti che insieme formano una soluzione del CSP. Si noti che un CSP *fallito* è anche *inconsistente*, ma non tutti i CSP *inconsistenti* sono *falliti*.

Siano $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ e $P' = (\mathcal{X}', \mathcal{D}', \mathcal{C}')$ due CSP. P e P' sono detti **equivalenti** se hanno lo stesso insieme delle soluzioni. P è detto *minore* di P' se sono equivalenti e $D_x \subseteq D'_x$ per tutti gli $x \in \mathcal{X}$ e viene indicato $P \preceq P'$. P è *strettamente minore* di P' se $P \preceq P'$ e $D_x \subset D'_x$ per almeno un $x \in \mathcal{X}$ e

viene indicato con $P \prec P'$. Se $P \preceq P'$ e $P' \preceq P$ scriveremo $P \equiv P'$.

2.2.2 Processo di soluzione

Nella programmazione a vincoli, il goal è quello di trovare una soluzione (o tutte le soluzioni) di un dato CSP. Il processo di soluzione prevede **ricerca**, **riduzione dei domini** e **propagazione di vincoli**.

La **ricerca** di una soluzione in un CSP i fatta spezzando iterativamente il CSP in CSP più piccoli. Questo processo di frammentazione è detto costruzione dell'**albero di ricerca**. Un nodo dell'albero corrisponde a un CSP. Inizialmente alla radice avremo un CSP P_0 che modella il problema che vogliamo risolvere. Se P_0 non è ne risolto ne fallito, dividiamo P_0 in più CSP $P_1, P_2, \ldots, P_k$ $(k > 1)$ assicurandoci che tutte le soluzioni di P_0 siano preservate e che nessuna nuova soluzione sia aggiunta e cioè che l'unione delle soluzioni di $P_1, P_2, \ldots, P_k$ sia uguale all'insieme delle soluzioni di P_0. In più ogni CSP $P_i (i > 0)$ deve essere strettamente minore di P_0 per assicurarci che il processo di frammentazione abbia fine. Successivamente ogni CSP $P_1, P_2, \ldots, P_k$ viene suddiviso con lo stesso procedimento di cui sopra. Il procedimento continua fino ad ottenere CSP risolti o falliti. Questi sono le foglie dell'albero di ricerca.

La dimensione dell'albero di ricerca cresce esponenzialmente con il numero delle variabili di P_0. Per ridurre questa dimensione la programmazione a vincoli utilizza un processo chiamato **propagazione di vincoli**. Dato un vincolo C e una nozione di *consistenza locale*, un **algoritmo di riduzione dei domini** rimuove valori che non sono consistenti con C dai domini delle variabili in C. L'algoritmo deve preservare tutte le soluzioni e non aggiungere nessuna soluzione a C. Se un valore inconsistente viene rimosso da un dominio di una variabile, l'effetto viene **propagato** a tutti gli altri vincoli che condividono quella variabile. Questo processo iterativo continua fino a che non è rimasto più nessun valore inconsistente in nessun vincolo del CSP.

Arrivati a questo punto diremo che il CSP è *localmente consistente* e cioè che il CSP non è *globalmente consistente* ma che ogni vincolo del CSP è individualmente (localmente) consistente.

Dopo aver diviso un CSP, i processi di riduzione dei domini e propagazione di vincoli vengono applicati ai CSP ottenuti. La rimozione di valori dai domini porta ad un albero delle soluzioni più piccolo e per questo velocizza il processo di soluzione. Dall'altra parte, è da mettere in conto il tempo speso nella propagazione di vincoli, per questo è necessario applicare algoritmi efficienti. L'efficienza dell'algoritmo è spesso determinato dalla nozioni di consistenza applicate ai vincoli. Rimane il problema di decidere in quali casi applicare una nozione anziché altre.

2.2.3 Nozioni di consistenza locale

Di seguito richiamiamo quattro definizioni di consistenza locale.

Definizione 2.1 (Arc consistency) *Un vincolo binario $C(x_1, x_2)$ è arc consistent se e solo se per ogni valore $d_1 \in D_{x_1}$ esiste un valore $d_2 \in D_{x_2}$ tale che $(d_1, d_2) \in C$, e per ogni valore $d_2 \in D_{x_2}$ esiste un valore $d_1 \in D_{x_1}$ tale che $(d_2, d_1) \in C$.*

Definizione 2.2 (Hyper-arc consistency) *Un vincolo $C(x_1, x_2, \ldots, x_m)$ $(m > 1)$ è hyper-arc consistent se e solo se per ogni $i \in \{1, 2, \ldots, m\}$ e per ogni valore $d_i \in D_{x_i}$ esistono valori $d_j \in D_{x_j}$ per tutti $j \in \{1, 2, \ldots, m\} - i$ tali che $(d_1, d_2, \ldots, d_m) \in C$.*

Si noti che l'arc consistency è uguale all'hyper-arc consistency applicata a un vincolo binario. Entrambe, arc consistency e hyper-arc consistency, assicurano che tutti i valori di ogni dominio appartengano a tuple che soddisfano il vincolo.

Le due seguenti nozioni di consistenza locale considerano gli estremi dei domini delle variabili. Per questa ragione assumiamo che i domini delle

variabili coinvolte siano sottoinsiemi, su cui valori è definita una relazione d'ordine, di insiemi finiti. Sia D un dominio, definiamo $\max(D)$ e $\min(D)$, rispettivamente, come il suo massimo e minimo valore. Caso particolare di questi domini sono gli intervalli dove sono presenti tutti i valori tra $\max(D)$ e $\min(D)$. Le parentesi { } e [ ] saranno usate per indicare insiemi e intervalli rispettivamente. Ad esempio l'insieme {1,3} contiene i valori 1 e 3, mentre l'intervallo $[1..3] \subset \mathbb{Z}$ contiene 1,2 e 3.

Definizione 2.3 (Bound consistency) *Un vincolo* $C(x_1, x_2, \ldots, x_m)$ $(m > 1)$ *è bound consistent se e solo se per ogni* $i \in \{1, 2, \ldots, m\}$ *e per ogni valore* $d_i \in \{\min(D_{x_i}), \max(D_{x_i})\}$ *esistono valori* $d_j \in [\min(D_{x_j})..\max(D_{x_j})]$ *per tutti* $j \in \{1, 2, \ldots, m\} - i$ *tali che* $(d_1, d_2, \ldots, d_m) \in C$.

Definizione 2.4 (Range consistency) *Un vincolo* $C(x_1, x_2, \ldots, x_m)$ $(m > 1)$ *è range consistent se e solo se per ogni* $i \in \{1, 2, \ldots, m\}$ *e per ogni valore* $d_i \in D_{x_i}$ *esistono valori* $d_j \in [\min(D_{x_j})..\max(D_{x_j})]$ *per tutti* $j \in \{1, 2, \ldots, m\} - i$ *tali che* $(d_1, d_2, \ldots, d_m) \in C$.

Queste due definizioni di consistenza non considerano esattamente i valori del dominio ma il più piccolo intervallo che lo contiene. Per questa ragione range consistency e bound consistency non garantiscono l'esistenza di una soluzione per il vincolo mentre l'hyper-arc consistency sì. Questo è formalizzato qui di seguito.

Definizione 2.5 (Consistenza Locale CSP) *Un CSP è rispettivamente range consistent, bound consistent o hyper-arc consistent se tutti i suoi vincoli lo sono.*

Se applichiamo ad un CSP P un algoritmo di propagazione che stabilisce range consistency su P, denotiamo il risultato con $\Phi_R(P)$. Analogamente, $\Phi_B(P), \Phi_A(P)$ e $\Phi_{HA}(P)$ denotano l'applicazione di bound consistency, arc consistency e hyper-arc consistency su P, rispettivamente.

Teorema 2.1 *Sia P un CSP, allora $\Phi_{HA}(P) \preceq \Phi_R(P) \preceq \Phi_B(P)$.*

Dimostrazione. Se un vincolo è hyper-arc consistent allora è anche range consistent dal momento che $D_x \subseteq [\min(D_x)..\max(D_x)]$ per tutte le variabili x in P. Il contrario non è vero quindi $\Phi_{HA}(P) \preceq \Phi_R(P)$ potrebbe essere stretto.

Se un vincolo è range consistent allora è anche bound consistent dal momento che $\{\min(D_x), \max(D_x\} \subseteq D_x$ per tutte le variabili x in P. Il contrario non è vero quindi $\Phi_R(P) \preceq \Phi_B(P)$ potrebbe essere stretto. □

Capitolo 3

Risolutore di Problemi a Vincoli su Domini Finiti

In questo capitolo è descritta la teoria su cui si basa la progettazione e l'implementazione del risolutore di problemi a vincoli su domini finiti JSetL($\mathcal{FD}$).

3.1 Vincoli $\mathcal{FD}$ e Domini

Il linguaggio degli $\mathcal{FD}$-*constraint* si basa sul seguente alfabeto [12]:

- un insieme F di costanti e simboli di funzione per rappresentare gli elementi di $\mathbb{Z}$ $(0, -1, 1, -2, 2, \ldots)$ e un insieme di simboli di funzione per rappresentare gli operatori aritmetici standard $+, -, \cdot, /$;
- un insieme di simboli predicato $\Pi_c = \{\in [a..b], =, \neq, <, \leq, >, \geq\}$;
- un insieme numerabile di variabili V (solitamente indicate con le lettere $x, y, z, \ldots$).

Definizione 3.1 (Vincolo $\mathcal{FD}$ primitivo) *Un vincolo $\mathcal{FD}$ primitivo è qualsiasi predicato atomico costruito con i simboli dell'alfabeto degli $\mathcal{FD}$-*constraint.

Esempi di vincoli $\mathcal{FD}$ primitivi sono:

- $x \in [1..10]$
- $x < 4$
- $y \geq x$
- $x \cdot y \neq z + 10$

Definizione 3.2 (Vincolo $\mathcal{FD}$) *Un vincolo $\mathcal{FD}$ (non primitivo) è una congiunzione di vincoli $\mathcal{FD}$ primitivi.*

Esempi di vincoli $\mathcal{FD}$ (non primitivi) sono:

- $x \in [1..10],\ x < 6,\ x \geq y$
- $z \geq x,\ z \in [12..20],\ x \cdot y \neq z + 10$

Altri vincoli primitivi sono spesso forniti nei linguaggi di programmazione logica a vincoli. Ad esempio il vincolo *globale* `allDifferent` che verrà trattato singolarmente nel capitolo 6.

L'interpretazione di questo linguaggio è quella naturale che mappa i simboli dell'alfabeto in elementi di $\mathbb{Z}$ e in funzioni e relazioni su $\mathbb{Z}$ nel modo comunemente conosciuto. In modo particolare, l'interpretazione del vincolo $x \in [a..b]$ è vera *se e solo se* $a \leq x \leq b$.

I vincoli del tipo $\in [a..b]$ sono utilizzati per identificare intervalli rappresentanti i domini delle variabili del problema.

Il dominio di una variabile è rappresentato come un intervallo di interi. L'intervallo è specificato dai sui estremi, destro e sinistro. Il dominio D_x associato a una variabile x può essere solo $D_x = [a..b]$ con a e b interi e $a \leq b$. Un dominio è detto **singoletto** nel caso particolare in cui sia $a = b$ è cioè $D_x = [a..a]$ che vale anche a dire: il dominio di x contiene l'unico valore intero a oppure il $x = a$. Diremo, invece, che $D_x = \emptyset$ (insieme vuoto) se $a > b$.

3.2 Regole di Riduzione dei Domini

In questo paragrafo verranno esaminati singolarmente i vincoli $\mathcal{FD}$ e per ognuno verrà data una **regola di riduzione dei domini**, e cioè, quali valori sono da eliminare dal dominio delle variabili coinvolte per far sì che la nozione di consistenza locale del vincolo sia rispettata.

Ad ogni vincolo sarà, quindi, anche associata una nozione di consistenza locale. Nel fare questa scelta si deve considerare da una parte il *potere riduttivo* sui domini e, dall'altro, il costo computazionale e la fattibilità della implementazione. Se non specificato la nozione di consistenza locale utilizzata sarà *hyper-arc consistent*.

In generale si procede in questo modo. Per ogni variabile coinvolta si determina quale insieme di valori in $\mathbb{Z}$ soddisfa il vincolo e poi questo insieme viene intersecato con il dominio della variabile stessa. Agendo per intersezione garantiamo che il processo di riduzione non aggiunga nessuna soluzione al vincolo.

Intersezione di Intervalli. L'intersezione di intervalli è sempre usata nelle regole che vedremo di seguito. Per questo è opportuno vedere come è calcolata.

$$[a..b] \cap [c..d] = [\max(a, b).. \min(c, d)]$$

.

3.2.1 Vincolo di Dominio

Il vincolo di dominio viene rappresentato con l'operatore $\in (.)$.

Regola 3.1 (Dominio) *Sia* $x \in [l_x..h_x]$. *L'applicazione del vincolo* $x \in ([a..b])$ *determina una restrizione del dominio di* x *tale che:*

$$x \in ([l_x..h_x] \cap [a..b])$$

3.2.2 Vincolo di Uguaglianza

Il vincolo binario di uguaglianza è rappresentato dal simbolo $=$. Similmente ad altri vincoli binari, il vincolo di uguaglianza può coinvolgere un'unica variabile nel caso $x = k$ con k costante intera, oppure può coinvolgerne due nel caso $x = y$. Il primo caso si può ricondurre al secondo con $y \in [k..k]$.

Regola 3.2 (Uguaglianza) *Sia $x \in [l_x..h_x]$ e $y \in [l_y..h_y]$. L'applicazione del vincolo $x = y$ determina una restrizione dei domini di x e y tale che:*

$$x, y \in ([l_x..h_x] \cap [l_y..h_y])$$

Si noti che l'applicazione del vincolo $x = y$, in termini di riduzione del dominio, equivale all'applicazione della coppia di vincoli

$$x\ in([l_y..h_y]),\ y\ in([l_x..h_x])$$

3.2.3 Vincoli di Disuguaglianza

Incominciamo col considerare che si possono ricondurre tutti i vincoli di disuguaglianza $\{<, \leq, >, \geq\}$ al solo vincolo $<$. Infatti:

- $x \leq y \equiv x < y + 1$
- $x > y \equiv y < x$
- $x \geq y \equiv x > y - 1 \equiv y < x + 1$

In realtà nella parte destra della prima e della terza equivalenza non si tratta più di semplici vincoli di disuguaglianza ma di vincoli aritmetici in quanto compare una espressione aritmetica. In questi casi si opera una sostituzione utilizzando una variabile ausiliaria e si riscrive il vincolo, ad esempio $x < y-1$, come la composizione di un vincolo di disuguaglianza, $x < z$, e di un vincolo aritmetico, $z = y - 1$. Questo ultimo tipo di vincolo è trattato nel paragrafo 3.2.4.

Prima di dare la regola di riduzione per il vincolo '$<$' richiamiamo la nozione di hyper-arc consistency (che coincide con la nozione di arc consistency trattandosi di vincolo binario) per il vincolo di disuguaglianza. Siano x, y due variabili con dominio associato rispettivamente D_x e D_y; la hyper-arc consistency per il vincolo $x < y$ è rispettata se

$$\forall x \in D_x(\exists y \in Y \mid x < y), \forall y \in Y(\exists x \in X \mid x < y)$$

Procediamo definendo la regola di riduzione del domino per il vincolo $x < y$.

Regola 3.3 (Disuguaglianza *minore di*) *Sia $x \in D_x$ e $y \in D_y$ e denominiamo, per comodità, $\min(D_x) := l_x, \max(D_x) := h_x, \min(D_y) := l_y$ e $\max(D_y) := h_y$. L'applicazione del vincolo $x < y$ determina*

$$x \in [l_x .. \min(h_x, h_y - 1)]$$

$$y \in [\max(l_x + 1, l_y) .. h_y]$$

Proposizione 3.1 *La regola di riduzione dei domini 3.3 rispetta la nozione di hyper-arc consistency.*

Dimostrazione. Siano x, y variabili e siano rispettivamente $D_x = [l_x..h_x]$ e $D_y = [l_y..h_y]$ i loro domini associati. $x \in [l_x .. \min(h_x, h_y - 1)]$, allora
$\forall x(x \leq \min(h_x, h_y - 1))$, siccome $\min(h_x, h_y - 1) < h_y$, $\forall x(x < h_y)$, allora
$\forall x \in D_x(\exists y \in D_y \mid x < y)$ $y \in [\max(l_x + 1, l_y)..h_y]$, allora
$\forall y(y \geq \max(l_x + 1, l_y))$, siccome $\max(l_x + 1, l_y) > l_x$, $\forall y(y > l_x)$, allora
$\forall y \in D_y(\exists x \in D_x \mid x < y)$ □

3.2.4 Vincoli Aritmetici

I vincoli aritmetici sono vincoli che mettono a confronto due espressioni aritmetiche definite su interi. Sono quindi della forma t *op* s con t ed s espressioni aritmetiche e $op \in \{<, \leq, =, \neq, >, \geq\}$. Ad esempio

Esempio 3.1

$$2 + 4 \cdot x^3 \cdot 7 + z^6 \leq y - 9 + x$$

è un vincolo aritmetico. Qui x^3 è un'abbreviazione di $x \cdot x \cdot x$ e similmente le altre espressioni, mentre $y - 9$ abbrevia la somma tra la variabile y e l'intero -9.
Introduciamo ora i **vincoli aritmetici atomici**. I vincoli aritmetici atomici sono quelli della forma

- $x \cdot y = z$, oppure
- $x + y = z$

e li chiameremo rispettivamente: **vincolo di moltiplicazione** e **vincolo di somma**.

I vincoli atomici di divisione e sottrazione vengono ricondotti rispettivamente ai vincoli di moltiplicazione e somma. Nel caso in cui l'espressione aritmetica contenga un vincolo di divisione x/y, l'operazione di divisione verrà sostituita con una variabile v_i ausiliaria e verrà aggiunto un vincolo di moltiplicazione $x = y \cdot v_i$ al problema. Similmente avviene per la sottrazione.

Riprendendo l'esempio di sopra, si può riscrivere.

$$2 + 4 \cdot x^3 \cdot 7 + z^6 \leq y - 9 + x$$

come la serie di vincoli

$$2 + v_1 \cdot x^2 \cdot 7 + z^6 \leq v_2 + x,$$

$$v_1 = 4 \cdot x,\ v_2 = y - 9$$

e continuando così

$$2 + v_3 \cdot x \cdot 7 + z^6 \leq v_4$$

$$v_1 = 4 \cdot x,\ v_2 = y - 9,\ v_3 = v_1 \cdot x,\ v_4 = v_2 + x$$

È facile vedere come con l'aggiunta di variabili ausiliarie sia possibile riscrivere un vincolo aritmetico come una serie di vincoli aritmetici atomici e al più un vincolo di disuguaglianza o un vincolo di uguaglianza o di diverso. Quindi nel seguito considereremo soltanto il trattamento dei vincoli atomici.

3.2.4.1 Vincolo di Somma

Vincolo di somma

$$x + y = z$$

in presenza di domini non vuoti $x \in D_x,\ y \in D_y$ e $z \in D_z$.
Prima di tutto estendiamo le definizioni di operazioni somma e sottrazione sul dominio degli intervalli di interi e poi di seguito diamo la regola di restrizione di dominio dovuto al vincolo $x + y = z$ per le variabili x, y, z.

Definizione 3.3 (Somma di insiemi di interi) *Dati due insiemi di interi X,Y, definiamo la loro somma come segue*

$$X + Y := \{x + y \mid \exists x \in X, \exists y \in Y\}$$

Definizione 3.4 (Sottrazione di insiemi di interi) *Dati due insiemi di interi Z,Y,definiamo la loro differenza come segue*

$$Z - Y := \{x \in \mathbb{Z} \mid \exists y \in Y \exists z \in Z x + y = z\}$$

Proposizione 3.2 *Siano $x \in D_x$ e $y \in D_y$:*

$$x + y \in [\min(D_x) + \min(D_y) .. \max(D_x) + \max(D_y)]$$

Dimostrazione. Sia $x \in X, y \in Y$

1. $\forall x, y(x \geq \min(D_x), y \geq \min(D_y))$, allora
   $\forall x, y(x + y \geq \min(D_x) + \min(D_y))$

2. $\forall x, y(x \leq \max(D_x), y \leq \max(D_y))$, allora
   $\forall x, y(x + y \leq \max(D_x) + \max(D_y))$,

allora per 1 e 2

$$\forall x, y(\min(D_x) + \min(D_y) \leq x + y \leq \max(D_x) + \max(D_y)$$

allora

$$x + y \in [\min(D_x) + \min(D_y) .. \max(D_x) + \max(D_y)]$$

□

Allo stesso modo se $x \in D_x, z \in D_z$:

$$x - z \in [\min(D_x) - \max(D_z) .. \max(D_x) - \min(D_z)]$$

. La dimostrazione segue dalla precedente.

Regola 3.4 (Somma) *Siano $x \in [l_x..h_x]$, $y \in [l_y..h_y]$ e $z \in [l_z..h_z]$; l'applicazione del vincolo $x + y = z$ determina una riduzione dei domini tale che:*

- $x \in D_x \cap [(l_z - h_y)..(h_z - l_y)]$,
- $y \in D_y \cap [(l_z - h_x)..(h_z - l_x)]$,
- $z \in D_z \cap [(l_x + l_y)..(h_x + h_y)])$

3.2.4.2 Vincolo di Moltiplicazione

Vediamo nello specifico il vincolo di moltiplicazione

$$x \cdot y = z$$

in presenza di domini non vuoti $x \in D_x$, $y \in D_y$ e $z \in D_z$. Come fatto per la somma è utile estendere l'operazione di moltiplicazione agli intervalli di interi o più generalmente agli insiemi di interi.

Definizione 3.5 (Moltiplicazione di insiemi di interi) *Dati due insiemi di interi X,Y, definiamo la loro moltiplicazione come segue*

$$X \cdot Y := \{x \cdot y \mid x \in X, y \in Y\}$$

Notiamo che per due intervalli di interi X, Y la loro moltiplicazione non è necessariamente un intervallo. Per esempio

$$[0..2] \cdot [1..2] = \{x \cdot y \mid x \in [0..2], y \in [1..2]\} = \{0, 1, 2, 4\}$$

Di conseguenza, per mantenere la proprietà che i domini su variabili sono intervalli di interi, introduciamo il seguente operatore sui sottoinsiemi dell'insieme degli interi $\mathbb{Z}$:

Definizione 3.6 (Operatore di chiusura su intervallo)

$$int(X) := \begin{cases} \textit{il più piccolo intervallo che contiene } X & \textit{se } X \textit{ è finito} \\ \mathbb{Z} & \textit{altrimenti} \end{cases}$$

Quindi per definizione $int([0..2] \cdot [1..2]) = [0..4]$. Il vincolo di moltiplicazione è quindi *bound consistent* in quanto solo gli estremi del dominio devono rispettare il vincolo.

Per poter trattare la riduzione dei domini di x e y rispetto al vincolo $x \cdot y = z$ è necessario introdurre la divisione di intervalli di interi. Una complicazione è che dobbiamo considerare la divisione per zero.

Definizione 3.7 (Divisione di insiemi di interi) *Dati due insiemi di interi Z,Y definiamo la loro divisione come segue*

$$Z/Y := \{x \in \mathbb{Z} \mid \exists y \in Y \ \exists z \in Z \ x \cdot y = z\}$$

Di nuovo, come per la moltiplicazione, la divisione di due intervalli non è necessariamente un intervallo. Ad esempio

$$[3..5]/[-1..2] = \{-5, -4, -3, 2, 3, 4, 5\}$$

e quindi ci serviremo ancora dell'operatore $int(.)$: $int([3..5]/[-1..2]) = [-5..5]$. Si noti anche che se $0 \in Y \cap Z$, allora $Z/Y = \mathbb{Z}$. Ed ancora, se $0 \notin Z$, allora $Z/[0..0] = \emptyset$.

Siamo pronti ora a trattare il vincolo di moltiplicazione.

Regola 3.5 (Moltiplicazione) *Siano $x \in D_x$, $y \in D_y$ e $z \in Dz$; l'applicazione del vincolo $x \cdot y = z$ determina*

- $x \in D_x \cap int(D_z/D_y)$,
- $y \in D_y \cap int(D_z/D_x)$,
- $z \in D_z \cap int(D_x \cdot D_y)$

Vediamo ora come è calcolata la chiusura $int(.)$ sulla moltiplicazione e la divisione di intervalli. Per quanto riguarda la moltiplicazione il calcolo è piuttosto semplice. Siano a, b, c, d interi con $a \leq b$ e $c \leq d$ e sia $A = \{a \cdot b, a \cdot d, b \cdot c, b \cdot d\}$

$$int([a..b] \cdot [c..d]) = [\min(A).. \max(A)]$$

Per quanto riguarda la divisione invece è necessario distinguere diversi casi.

1. Supponiamo $0 \in [a..b]$ e $0 \in [c..d]$.
   Allora per definizione $int([a..b]/[c..d]) = \mathbb{Z}$. Ad esempio
   $$int([-10..2]/[-1..4]) = \mathbb{Z}.$$
2. Supponiamo $0 \notin [a..b]$ e $c = d = 0$.
   Allora per definizione $int([a..b]/[c..d]) = 0$. Ad esempio
   $$int([1..10]/[0..0]) = 0.$$
3. Supponiamo $0 \notin [a..b]$ e $c < 0 < d$.
   È facile vedere che $int([a..b]/[c..d]) = [-e..e]$, dove $e = \max(|a|, |b|)$. Ad esempio
   $$int([-19.. -1][-3..5]) = [-19..19].$$

4. Supponiamo $0 \notin [a..b]$ e $c = 0, d \neq 0$ oppure $c \neq 0, d = 0$.
   Allora $int([a..b]/[c..d]) = int([a..b]/([c..d])\backslash 0)$. Ad esempio
   $$int([1..10]/[-4..0]) = int([1..10]/[-4..-1]).$$

5. Supponiamo $0 \notin [c..d]$
   Allora $int([a..b]/[c..d]) = [\lceil \min(A) \rceil .. \lfloor \max(A) \rfloor]$.
   Dove $A = \{a/c, a/d, b/c, b/d\}$. Ad esempio
   $$int([-10..4]/[3..4]) = [-3..1]$$

3.2.5 Vincolo di Diverso

Osserviamo come la nozione di hyper-arc consistecy per il vincolo $x \neq y$ è rispettata semplicemente se

$$\exists d_1 \in D_x, \exists d_2 \in D_y \mid d_1 \neq d_2$$

e cioè è sufficiente che esista una coppia di valori per x, y tali che $x \neq y$ perché il vincolo venga rispettato; il che vuol dire che se $\mid D_x \mid, \mid D_y \mid > 1$, cioè entrambi i domini contengono più di un valore, allora la nozione di consistenza locale sul vincolo è sempre rispettata e nessuna riduzione dei domini può essere fatta. Mentre se i domini delle variabili coinvolte sono singoletti nessuna riduzione di dominio è necessaria per determinare la consistenza del vincolo. Per poter fare considerazioni sui domini è necessario che una delle due variabili abbia un dominio singoletto, un unico valore possibile.
Vediamo ad esempio il caso in cui $y \in [k..k]$ con $k \in \mathbb{Z}$. Possiamo prevedere tre casi per $x \neq k$:

1. $k \notin D_x$, il vincolo di diverso è già consistente senza nessuna riduzione di dominio.

2. $k = \min(D_x)$ oppure $k = \max(D_x)$, il dominio di x viene ristretto a $D_x\backslash\{k\}$. Si noti che se $D_x = [k..k]$ allora dalla regola di riduzione si ha dominio $D_x = \emptyset$ e cioè che il vincolo è inconsistente.

3. $\min(D_x) < k < \max(D_x)$, per necessità viene applicata la nozione di bound consistency e non si ha nessuna riduzione di dominio. Non possiamo eliminare il valore k dal dominio di x altrimenti perderemmo la proprietà per cui tutti i domini sono intervalli. Anche se applicassimo l'operatore $int(.)$, definito per il vincolo aritmetico di moltiplicazione, sarebbe ininfluente in quanto in questo caso $int(D_x/k) = int([\min(D_x)..k-1] \cup [k+1..\max(D_x)]) = D_x$.

Abbiamo visto come nel terzo caso non si abbia molta possibilità di ridurre i domini delle variabili coinvolte e che la nozione di consistenza utilizzata è meno stringente, non più hyper-arc consistency ma bound consistency. La programmazione a vincoli ci da però la possibilità di *spezzare* un problema a vincoli in due sotto problemi.

Consideriamo il CSP P con il solo vincolo $C := x \neq k$, e sia $\min(D_x) < k < \max(D_x)$. Spezziamo ora il CSP P in due CSP P_1 e P_2, dove P_1 è un CSP che ha come unico vincolo $C_1 := x < k$ e P_2 come unico vincolo $C_2 := k < x$. In questo modo è possibile applicare la regola di riduzione dei domini per il vincolo *minore di* ai due CSP figli. Nella dimostrazione seguente è mostrato che eseguire la riduzione dei domini con questo approccio non aggiunge ne toglie soluzioni al CSP padre.

Proposizione 3.3 *Siano P, P_1 e P_2 tre CSP, e supponiamo che P abbia come unico vincolo $C := x \neq k$, che P_1 abbia come unico vincolo $C_1 := x < k$, che P_2 abbia come unico vincolo $C_2 := k > x$, che sia $D_x := [l_x..h_x]$ e che sia $l_x < k < h_x$.*
L'unione degli insiemi delle soluzioni di P_1 e P_2 equivale all'insieme delle soluzioni di P.

Dimostrazione. Applicando la nozione di hyper-arc consistency sui vincoli dei problemi P, P_1, e P_2:

- per C si ha una riduzione dei domini tale che $x \in (D_x \backslash \{k\})$,

- per C_1 si ha (per la regola 3.3) $x \in [l_x .. \min(k-1, h_x)]$ e siccome $k < h_x,\ x \in [l_x..k-1]$

- per C_2 si ha (per la regola 3.3) $x \in [\max(k+1, l_x)..h_x]$ e siccome $k > l_x,\ x \in [k+1..h_x]$

essendo $(D_x \backslash \{k\}) = ([l_x..k-1] \cup [k+1..h_x])$ l'unione degli insiemi delle soluzioni di P_1 e P_2 è uguale all'insieme delle soluzioni di P. □

3.3 Propagazione dei Vincoli

La propagazione dei vincoli è ottenuta in modo molto semplice e naturale. L'obbiettivo della propagazione è che gli effetti di ogni vincolo si ripercuota sugli altri vincoli presenti nel problema.

Il risolutore detto tecnicamente **solver** considera sequenzialmente tutti i vincoli che compongono il problema. Chiamiamo ora C_2 il vincolo esaminato per secondo. Le variabili in C_2 avranno subito le eventuali riduzioni dal vincolo C_1(se coinvolte). Questa situazione si descrive come propagazione del vincolo C_1 a C_2.

Il problema è che il vincolo C_2 non si è propagato a C_1. Il discorso si può espandere in modo generale a tutti gli n vincoli del problema. La soluzione adottata è che risolto l'ultimo vincolo il solver riprende a considerare la sequenza di vincoli dal principio. Questo procedimento continua fino a quando si ha una scansione completa senza nessuna riduzione di dominio (punto fisso).

Prendiamo ad esempio il CSP $x < y, y < 3$ con $D_x = [1..3], D_y = [1..3]$. Il primo vincolo $x < y$ riduce i domini a: $D_x = [1..2], D_y = [2..3]$ e il secondo vincolo $y < 3$ riduce ulteriormente il dominio $D_y = [2..2]$ assegnando di fatto $y = 2$. A questo punto il risolutore riprende la sequenza di vincoli dal principio. Si trova quindi di fronte al CSP $x < y, y < 3$ con $D_x = [1..2], D_y = [2..2]$. Il primo vincolo $x < y$ riduce il dominio assegnando di fatto $x = 1$. La consistenza è verificata sul secondo vincolo in quanto $1 < 2$. A questo punto tutti i vincoli sono consistenti e tutte le variabili hanno un valore assegnata. Le regole di riduzione dei dominio e la propagazione dei vincoli hanno portato alla soluzione del problema: $x = 1, y = 2$.

Se durante la propagazione, uno dei vincoli diviene inconsistente o fallisce (variabile con dominio vuoto) allora il problema rappresentato dalla specificata terna di insiemi variabili, domini e vincoli non ha soluzione.

Vediamo ad esempio il CSP $x < y, y < x$ con $D_x = [1..4], D_y = [1..4]$. Il risolutore procede con il primo ciclo dove $x < y$ riduce $D_x = [1..3], D_y = [2..4]$ e $y < x$ riduce ulteriormente $D_x = [3..3], D_y = [2..2]$ assegnando di fatto $x = 3, y = 2$. Il secondo ciclo di risoluzione non verifica la consistenza del primo vincolo $x < y$ poiché 3 *non è* minore di 2. Di conseguenza il CSP non ha soluzione.

3.4 Consistenza globale e Ricerca della Soluzione

Tramite la riduzione dei domini non è sempre possibile arrivare ad una soluzione anche se il problema soddisfa la nozione di consistenza. Una soluzione si ha infatti solo quando i domini delle variabili coinvolte nel problema hanno tutte un valore assegnato (hanno cioè un dominio singoletto) e tutti i vincoli sono rispettati (consistenti). Ad esempio, il semplice problema $x < y$ con $D_x = D_y = [1..10]$, può essere reso consistente riducendo i domini a

$D_x = [1..9],\ D_y = [2..10]$ ma non si arriva a dare una soluzione (una soluzione è per esempio $x = 4,\ y = 7$).

Inoltre un problema a vincoli che è *(localmente) consistente* non è detto che abbia soluzione. Perché la soluzione esista è infatti necessario che la consistenza sia **globale**. Prendiamo ad esempio un CSP che ha come vincoli $x \neq y,\ x \neq z,\ y \neq z$ e $D_x = D_y = D_z = \{1, 2\}$. Ognuno dei 3 vincoli è consistente localmente e nessuna riduzione dei domini è necessaria. Tuttavia il problema non ha soluzione.

Come già detto nel paragrafo 2.2.2 il processo di soluzione prevede, una volta ridotti i domini e fatta la propagazione dei vincoli, di spezzare il CSP in un insieme di CSP più "semplici" per cercare la soluzione. Facciamo questo attraverso il processo di *labeling*.

3.4.1 Labeling

Fare il labeling significa semplicemente assegnare ad una variabile un valore del proprio dominio. Più interessante è vedere come il labeling partecipa al processo di ricerca della soluzione costruendone di fatto l'albero di ricerca.

Prendiamo il CSP P dove i domini sono già stati opportunamente ridotti secondo la nozione di consistenza richiesta, ma ancora non si sia arrivati alla soluzione ne a un fallimento. Esistono quindi alcune variabili senza un valore assegnato, ma con dominio non vuoto. Diciamo che una di queste variabili sia x con dominio $[l_x..h_x]$. P può essere spezzato in due CSP figli $P_{1,1}$ e $P_{1,2}$ dove in $P_{1,1}$ viene assegnato a x il valore l_x e in $P_{1,2}$ viene ridotto il dominio di x all'intervallo $[l_x + 1..h_x]$. Così facendo otteniamo due CSP il cui insieme delle soluzioni è uguale all'insieme delle soluzioni di P. La restrizione che si ha del dominio di x in ognuno dei CSP figli si propaga a tutti i vincoli innescando un nuovo processo di riduzione. È evidente che continuando con il labeling a spezzare iterativamente ogni CSP consistente ma non risolto, arriveremo prima o poi a una soluzione, se c'è. Una volta trovata la prima

soluzione possiamo decidere di fermarci o procedere e spezzare i CSP non ancora risolti in cerca di una ulteriore soluzione. Se si vogliono trovare tutte le soluzioni del problema bisogna spezzare tutti i CSP fino a che ognuna delle foglie dell'albero non sia un CSP risolto o fallito. Il problema P non ha soluzione se nessuna delle foglie del suo albero di ricerca della soluzione risulta essere un CSP risolto.

Alcuni sistemi permettono di specificare l'ordine di scelta delle variabili su cui fare labeling. L'ordine di scelta delle variabili viene solitamente determinato in base alle dimensioni dei domini ad esse associati e può essere crescente o decrescente.

Anche il modo in cui spezzare il dominio di ogni singola variabile può essere fatto con diverse tecniche. Una tecnica alternativa al labeling è la tecnica della **bisezione** che prevede di spezzare il dominio della variabile nel mezzo creando due CSP figli. Per maggiori informazioni su queste tecniche si veda [4].

L'ordine di scelta delle variabili e la tecnica di partizionamento dei domini possono portare ad un ricerca della soluzione più o meno veloce. Tuttavia non esistono strategie migliori di altre in assoluto: alcune possono essere più vantaggiose per un dato problema mentre altre per altri problemi. Nel nostro lavoro useremo la tecnica di labeling senza nessuna precedenza particolare sulle variabili.

Capitolo 4

JSetL($\mathcal{FD}$)

In questo capitolo viene prima data una breve introduzione alla libreria JSetL e poi vengono descritte quali potenzialità sono state introdotte nella versione JSetL($\mathcal{FD}$).

4.1 Una breve introduzione a JSetL

JSetL è una libreria Java che offre un supporto per la programmazione dichiarativa. Vediamone di seguito le caratteristiche principali.

- **Variabili logiche**. Le variabili logiche posso essere sia *inizializzate* che *non inizializzate*. Il valore di una variabile logica può essere di qualsiasi tipo. Un valore può essere assegnato ad una variabile solo se questa è *non inizializzata*.

- **Liste e Insiemi (logici)**. Queste strutture dati mantengono tutte le caratteristiche legate allo loro definizione classica ed in più possono essere parzialmente specificate. Essere una struttura parzialmente specificate significa avere uno o più elementi di tipo variabile logica non inizializzata o di tipo parzialmente specificato.

- **Vincoli**. I vincoli sono condizioni imposte sulle variabili logiche. Tramite l'imposizione dei vincoli vengono attribuiti valori a variabili non inizializzate o strutture parzialmente specificate vengono meglio definite.

- **Non Determinismo**. Non importa l'ordine in cui i vincoli vengono risolti per determinare la soluzione (o le soluzioni) del problema.

Esaminiamo in particolare alcuni aspetti essenziali relativi alla definizione e al trattamento dei vincoli in JSetL, il dominio dei vincoli è quello degli insiemi $\mathcal{SET}$ ereditariamente finiti (ovvero insiemi di oggetti qualsiasi che possono essere anche altri insiemi a loro volta finiti[12]), ampliato con alcuni semplici vincoli sugli interi.

Definizione 4.1 (Vincolo primitivo) *Un vincolo JSetL primitivo è una espressione di una delle seguenti forme:*

- `x.op();`
- `x.op(y);`
- `x.op(y,z);`

dove `op` *è uno dei* ***metodi predefiniti*** *per la gestione dei vincoli (*`eq, neq, in, nin, subset, union, size, lt, le, gt, ge` *ecc...) e* `x,y,z` *sono espressioni il cui tipo dipende dal vincolo* `op`*.*

Il significato di questi metodi è quello intuitivamente associato al loro nome: `eq` e `neq` sono vincoli di uguaglianza e disuguaglianza; `in` e `nin` sono vincoli di appartenenza e non appartenenza insiemistica; `subset` e `union` sono vincoli di inclusione e unione insiemistica; `lt, le, gt, ge` sono vincoli su interi corrispondenti rispettivamente alle relazioni $<, \leq, >, \geq$ e cosi via.

Definizione 4.2 (Vincolo) *Un vincolo JSetL è una congiunzione di uno o più vincoli primitivi, cioè un'espressione della forma:*

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

dove $c_1, c_2, \ldots, c_n$ *sono vincoli primitivi.*

Il significato di c_1`.and(`c_2`)....and(`c_n`)` è la **congiunzione logica**

Il **risolutore di vincoli** in JSetL è basato sulla riduzione di ogni vincolo primitivo in una forma semplificata, chiamata *forma risolta*, che è dimostrato essere **soddisfacibile**. Il successo di questa riduzione su tutti i vincoli primitivi del problema permette quindi di concludere che il problema è soddisfacibile. In caso contrario, l'individuazione di un *fallimento* (in termini logici, una riduzione a `false`) implica la non-soddisfacibilità del problema.

Una forma risolta ottenuta da un vincolo C rappresenta una **soluzione** per C. Un vincolo C può avere più di una soluzione: in questo caso il risolutore di JSetL cerca tutte le soluzioni di C in modo *non-deterministico* utilizzando strumenti come punti di scelta e backtracking (vedi paragrafo 5.1.1). Tuttavia, è anche possibile che C non abbia soluzione: in questo caso il processo di riduzione fallisce e viene sollevata l'eccezione `Failure`.

Diremo che una computazione **termina con fallimento** se essa causa la generazione dell'eccezione `Failure`; in caso contrario si dirà che la computazione **termina con successo**.

Quanto detto sopra è corretto soltanto nel caso in cui il problema non preveda vincoli aritmetici o tutti gli eventuali vincoli aritmetici possono essere scritti in **true**, o in vincoli di uguale o diverso. In caso contrario i vincoli aritmetici vengono lasciati irrisolti e non si è in grado quindi di determinare la consistenza del problema che in tal caso viene detto essere in *Weak solved form*. Questo perché JSetL non tratta vincoli aritmetici se tutti gli operatori non sono inizializzati. In JSetL la risoluzione un vincolo aritmetico con variabili non inizializzate viene rimandata affinché dalla propagazione di altri vincoli si possano determinare i valori di tutte le variabili coinvolte. Ovviamente ci sono molti casi dove questo non accade mai.

Per determinare la soddisfacibilità di problemi che si trovano in Weak solved form JSetL [17] utilizza alcune tecniche per verificare la consistenza globale dove il risolutore fa considerazioni sulla consistenza di più vincoli presi assieme. Ad esempio, applica la proprietà transitiva in modo da determinare l'inconsistenza del problema $x < 8,\ 8 < x$. Tuttavia queste tecniche risultano essere molto costose e di efficacia limitata. Ad esempio risulta difficile e pesante verificare la consistenza globale del problema $x > 2,\ y - 2 > 0,\ x + y > z,\ z < 4$.

Il **constraint store** di un risolutore S contiene la collezione di tutti i vincoli attualmente attivi in S. JSetL fornisce metodi attraverso i quali è possibile aggiungere nuovi vincoli al constraint store, visualizzarne il contenuto, rimuovere tutti i vincoli in esso presenti.

L'**inserimento** di un nuovo vincolo C al c.store è effettuato con la chiamata al metodo `add` della classe `SolverClass`: l'invocazione

$$S\texttt{.add(}C\texttt{)}$$

comporta l'aggiunta di C al constraint store del risolutore S. La collezione di vincoli contenuti nel constraint store è interpretata come una congiunzione di vincoli JSetL primitivi.

Dopo che i vincoli sono stati aggiunti al constraint store, è possibile richiedere al risolutore S di risolve il problema invocando il metodo `solve`:

$$S\texttt{.solve().}$$

Il metodo `solve` ricerca in modo non deterministico una soluzione che soddisfi tutti i vincoli presenti nel constraint store.

4.2 L'introduzione degli $\mathcal{FD}$ in JSetL

Il problema che si ha con la tecnica di risoluzione dei vincoli attualmente usata in JSetL è che l'albero di ricerca della soluzione generato è spesso troppo

ampio per poter essere esplorato in tempo ragionevole (per maggiori informazioni si veda [6]). Infatti in JSetL non viene applicata nessuna riduzione dei domini delle variabili ma viene utilizzata una strategia di ricerca della soluzione di tipo *generate & test*. Con questa tecnica viene prima generata una ipotetica soluzione assegnando ad ogni variabile uno dei possibili valori che ha nel proprio dominio e poi viene controllata la consistenza di ogni vincolo. Se poi ogni vincolo risulta consistente si eseguono alcuni test di consistenza globale altrimenti si ha un fallimento e si procede con il generare un'altra ipotetica soluzione. Ci si rende facilmente conto che al minimo aumentare dei domini delle variabili si ha un consistente aumento del tempo di ricerca della soluzione. Se, ad esempio, si vogliono determinare tutte le soluzione del problema del n *regine* visto nel paragrafo 2.1, per $n = 8$ si hanno $8^8 = 16\ 777\ 216$ ipotetiche soluzioni da generare e da verificare. Inoltre come detto sopra, il trattamento dei vincoli aritmetici in JSetL è molto parziale per cui spesso non si è in grado di determinarne la consistenza.

JSetL($\mathcal{FD}$) è un'*estensione* della libreria JSetL che prevede l'integrazione di un risolutore di vincoli $\mathcal{FD}$ all'interno del risolutore di vincoli insiemistici fornito da JSetL. Questo può essere fatto senza sostanziali modifiche al linguaggio di vincoli attualmente fornito da JSetL.

Vediamo i vantaggi offerti da JSetL($\mathcal{FD}$) rispetto a JSetL.

- Possibilità di trattare vincoli aritmetici e disuguaglianze su interi come veri vincoli, risolvibili anche quando parte degli operandi non hanno un valore noto.

- Maggiore efficenza di risoluzione di problemi che possono essere modellati come problemi a vincoli su domini finiti e che contengono vincoli aritmetici. La riduzione dei domini delle variabili del problema fornita dalle tecniche $\mathcal{FD}$ riduce lo spazio di ricerca della soluzione del problema stesso.

- Maggiore efficenza nella verifica della consistenza. Le informazioni sui domini delle variabili permettono la propagazione dei vincoli vista nel paragrafo 3.3. Senza l'utilizzo degli $\mathcal{FD}$ non è possibile stabilire l'inconsistenza di problemi del tipo $x < 3,\ x > 3$ quando x è non inizializzata a meno dell'utilizzo di tecniche per la verifica della consistenza globale che però risultano essere pesanti e in alcuni casi inefficaci.

- Maggiore efficacia e completezza nella verifica di vincoli di **cardinalità**, ad esempio il vicolo `size(S,n)` con `S` insieme logico e `n` variabile logica di tipo intero che impone $|S| = n$. Infatti per tali vincoli è richiesta la capacità di trattare sistemi relativamente complessi di equazioni e disequazioni su interi.
  Ad esempio il semplice problema $|S| = n,\ n < 0$ è chiaramente inconsistente perché la cardinalità di un insieme non può essere negativa. Tuttavia, senza le tecniche $\mathcal{FD}$, il vincolo $n < 0$ rimane irrisolto e non è possibile determinare l'inconsistenza del problema a meno dell'utilizzo di tecniche per la verifica della consistenza globale.

Nei paragrafi successivi vediamo quali funzioni sono state aggiunte alla estensione JSetL($\mathcal{FD}$) e ne spieghiamo il senso e l'utilizzo.

4.3 Vincolo di Dominio

Sia x una variabile logica (ovvero un oggetto di classe `Lvar`) e siano a e b espressioni di tipo intero. Un vincolo di dominio è un vincolo della forma

```
x.dom(a,b)
```

Il suo significato è quello di associare alla variabile x il dominio $[a..b]$. La semantica è quella di *in* cioè dell'appartenenza insiemistica di `x` all'insieme degli interi tra `a` e `b`, ma si è usato un metodo diverso per facilitarne il riconoscimento e il trattamento tramite tecniche $\mathcal{FD}$.

Vediamo alcuni esempi di utilizzo del vincolo di dominio e di come il vincolo è trattato dal risolutore.
Nel seguito supponiamo che `Solver` sia la specifica istanza della classe `SolverClass` dichiarata nel modo seguente

```
SolverClass Solver = new SolverClass();
```

e che le variabili logiche siano dichiarate nel modo seguente

```
Lvar x = new Lvar(x);
```

Es.1 Se la variabile logica di cui si specifica un dominio è non inizializzata il dominio viene semplicemente assegnato alla variabile.

```
//aggiunge al problema il vincolo x in [0..10]
Solver.add(x.dom(0, 10));
//esegue il processo di ricerca della soluzione
Solver.solve();


System.out.println(x);
```

Eseguendo il programma si ottiene il seguente output

```
>>Lvar: x, id: 0, is NOT initialized., domain:Int[0..10]
```

Es.2 Se la variabile di cui si specifica un dominio è inizializzata con un valore intero oppure ha già un dominio, alla variabile viene applicata la relativa regola di riduzione (regola 3.1). Qui sotto vediamo un esempio di problema consistente.

```
//aggiunge al problema la congiunzione di vincoli
//x in [0..10], y = 9
Solver.add(x.dom(0, 10).and(y.eq(9)));
```

```
//aggiunge al problema la congiunzione di vincoli
//x in [8..12], y in [8..12]
Solver.add(x.dom(8, 12).and(y.dom(8, 12)));

//esegue il processo di ricerca della soluzione
Solver.solve();

System.out.println(x);
System.out.println(y);
```

L'output prodotto sarà:

```
>>Lvar: x, id: 0, is NOT initialized., domain:Int[8..10]
>>Lvar: y, id: 1, val: 9
```

Es.3 Problema che non ha soluzione. Invece il problema seguente non ha soluzione. Infatti non può essere $y = 7$ e il suo dominio [8..10]

```
//aggiunge al problema il vincolo y = 7
Solver.add(y.eq(7));

//aggiunge al problema il vincolo y in [8..12]
Solver.add(y.dom(8, 12));

//esegue il processo di ricerca della soluzione
Solver.solve();
System.out.println(y);
```

L'output prodotto sarà:

```
>>Exception in thread "main" JSetL.Failure
```

Infatti non può essere `y = 7` e il suo dominio [8..12]

Es.4 Se la variabile di cui si specifica un dominio è inizializzata con un valore non intero si ha un fallimento.

```
Lvar Y = new Lvar("Y", 7.3);

//y in [0..10]
Solver.add(Y.dom(0, 10));

Solver.solve();

System.out.println(Y);
```

Eseguendo il programma si ottiene il seguente output:

```
>>Exception in thread "main" JSetL.Failure
```

4.4 Vincoli Aritmetici

Vediamo prima la sintassi e il significato dei vincoli aritmetici in JSetL e JSetL($\mathcal{FD}$). I vincoli aritmetici sono definiti solo su costanti intere o variabili a valore intero e sono della forma:

$$e_1.\texttt{op}(e_2)$$

dove `op` $\in$ {`eq, neq, lt, le, gt, ge`} (dove ogni operatore, mantenendo l'ordine, sta per $\{=, \neq, <, \leq, >, \geq\}$) ed e_1 e e_2 sono **espressioni aritmetiche intere** costruite con le funzioni `sum, mul, sub, div` (rispettivamente $+, \cdot, -, /$).

Se un vincolo aritmetico coinvolge variabili non inizializzate e senza dominio associato il risolutore considererà come sottinteso che tali variabili hanno dominio associato tutto l'insieme $\mathbb{Z}$.

Una variabile inizializzata con valore intero k è considerato un caso limite in cui il dominio di tale variabile è l'insieme contenete il solo valore k.

Se un vincolo aritmetico coinvolge variabili o una costanti di tipo non intero la risoluzione di tale vincolo genera un fallimento.

4.4.1 Vincolo di Uguaglianza

Il vincolo di uguaglianza in JSetL($\mathcal{FD}$) ha la forma generale

$$o_1\,.\,\texttt{eq}(o_2)$$

dove o_1, o_2 possono essere: espressioni aritmetiche oppure oggetti logici (variabile logica, insieme o lista). In generale la risoluzione di questo vincolo comporta l'unificazione tra o_1 e o_2. In particolare quando o_1 e o_2 sono espressioni aritmetiche il trattamento dell'uguaglianza comporta l'applicazione delle regole di riduzione dei domini viste nel capitolo precedente.
Vediamo qualche esempio del vincolo di uguaglianza su espressioni aritmetiche:

Es.1 Se $x = y$ e x e y sono due variabili logiche entrambe con dominio, le variabili verranno unificate e diventeranno di fatto la stessa variabile con dominio associato l'intersezione dei due domini (dalla regola di riduzione dei domini 3.2). Se questa intersezione è vuota allora si ha un fallimento. Ad esempio:

```
//aggiunge i vincoli di dominio
Solver.add(x.dom(0, 10));
Solver.add(y.dom(7, 20));

//aggiunge vincolo di uguaglianza
Solver.add(x.eq(y));
```

```
Solver.solve();


System.out.println(y);
System.out.println(x);
```

Eseguendo il programma si ottiene l'output:

```
Lvar: y, id: 1, is NOT initialized., domain:Int[7..10]
Lvar: y, id: 1, is NOT initialized., domain:Int[7..10]
```

Le due variabili vengono stampate con lo stesso nome perché in seguito all'unificazione sono diventate la stessa variabile.

Es.2 Se $x = y$ e x e y sono due variabili logiche di cui solo la prima ha un dominio associato mentre la seconda è non inizializzata, allora il sistema considererà come sottinteso che y ha dominio associato tutto l'insieme $\mathbb{Z}$. Dalla regola di riduzione del dominio 3.2 segue che il dominio associato alle due variabili dopo la loro unificazione sarà $D_x \cap \mathbb{Z}$ e quindi D_x. Ad esempio:

```
//aggiunge i vincoli di dominio
Solver.add(x.dom(0, 10));


//aggiunge vincolo il di uguaglianza x = y
Solver.add(x.eq(y));


Solver.solve();


System.out.println(y);
System.out.println(x);
```

Eseguendo il programma si ottiene l'output:

```
Lvar: x, id: 1, is NOT initialized., domain:Int[0..10]
Lvar: x, id: 1, is NOT initialized., domain:Int[0..10]
```

Es.3 Vediamo un esempio in cui si presenta l'espressione aritmetica $z = x + y$ dove le variabili hanno dominio associato rispettivamente $D_x = [0..20],\ D_y = [-6, 13]$ e $D_z = [2..7]$.

```
  //aggiunge i vincoli di dominio
  Solver.add(x.dom(0, 20));
  Solver.add(y.dom(-6, 13));
  Solver.add(z.dom(2, 7));

  //aggiunge vincolo z = x + y
  Solver.add(z.eq(y.sum(x)));
  Solver.solve();

System.out.println(x);
System.out.println(y);
System.out.println(z);
```

Dalla regola di riduzione di dominio 3.4 si ha che

- $x \in D_x \cap [(l_z - h_y)..(h_z - l_y)]$,
  $x \in ([2..7] \cap [(2 - 13)..(7 + 6)]),\ x \in ([2..7] \cap [-11..13])$,
  $x \in [2..7]$,
- $y \in D_y \cap [(l_z - h_x)..(h_z - l_x)]$,
  $y \in ([-6..13] \cap [(2 - 20)..(7 - 0)]),\ y \in ([-6..13] \cap [-18..7])$,
  $y \in [-6..7]$ e
- $z \in D_z \cap [(l_x + l_y)..(h_x + h_y)]$)
  $z \in ([2..7] \cap [(0 - 6)..(20 + 13)]),\ z \in ([2..7] \cap [-6..33])$,
  $z \in [2..7]$.

L'output prodotto:

```
Lvar: x, id: 0, is NOT initialized., domain:Int[2..7]
Lvar: y, id: 1, is NOT initialized., domain:Int[-6..7]
Lvar: z, id: 2, is NOT initialized., domain:Int[0..13]
```

mostra l'effetto della riduzione dei domini sulle tre variabili.

4.4.2 Vincolo di Disuguaglianza

Il vincolo di disuguaglianza è definito solo per espressioni aritmetiche di tipo intero. Ne consegue che una variabile o una costante di tipo non intero coinvolta in un vincolo di disuguaglianza produce un fallimento.

Vediamo la sintassi delle disuguaglianze in JSetL($\mathcal{FD}$) con qualche esempio:

- $y < x$
  `y.lt(x)`
- $z \leq x + (y \cdot 4)$
  `z.le(x.sum(y.mul(4)))`
- $y \geq x$
  `y.ge(x)`
- $z - 1 > (x/y)$
  `z.sub(1).gt(y.div(x))`

Vediamo ora un esempio più completo di definizione e trattamento di un vincolo di disuguaglianza in JSetL($\mathcal{FD}$). Di seguito il problema: $x+y < z+w$ con $D_x = [1..4],\ D_y = [1..3],\ D_z = [0..2]$ e $D_w = [-2..2]$.

```
//aggiunge i vincoli di dominio
Solver.add(x.dom(1, 4));
```

```
Solver.add(y.dom(1, 3));
Solver.add(z.dom(0, 2));
Solver.add(w.dom(-2, 2));

//aggiunge vincolo z = x + y
Solver.add(y.sum(x).lt(z.sum(w)));

Solver.solve();

System.out.println(x);
System.out.println(y);
System.out.println(z);
System.out.println(w);
```

In questo caso il sistema si serve di due variabili d'appoggio v_1 e v_2 in modo da dovere trattare solo vincoli aritmetici atomici, riscrivendo il vincolo $x + y < z + w$ come la congiunzione di vincoli:

$$v_1 = x + y,\ v_2 = z + w,\ v_1 < v_2$$

Dalla regola di riduzione dei domini 3.4 applicata sui due vincoli aritmetici avremo $v_1 \in [2..7]$ e $v_2 \in [-2..4]$. Una ulteriore riduzione dei domini si ha nell'applicare il vincolo di disuguaglianza $v_1 < v_2$ da cui (regola 3.3) $v_1 \in [2..3]$ e $v_2 \in [3..4]$. A questo punto il sistema applica la propagazione dei vincoli riconsiderando i due vincoli aritmetici $v_1 = x + y$ e $v_2 = z + w$ alla luce delle riduzioni sui domini fatte su v_1 e v_2. Il novo trattamento dei due vincoli aritmetici porta alle seguenti riduzioni $x \in [1..2]$, $y \in [1..2]$, $z \in [1..2]$ e $w \in [1..2]$. Nessun altra riduzione sui domini può essere determinata dai vincoli del problema.

E quindi l'output del programma sarà:

```
Lvar: x, id: 0, is NOT initialized., domain:Int[1..2]
Lvar: y, id: 1, is NOT initialized., domain:Int[1..2]
Lvar: z, id: 2, is NOT initialized., domain:Int[1..2]
Lvar: w, id: 3, is NOT initialized., domain:Int[1..2]
```

4.4.3 Vincolo di Diverso

Il vincolo di diverso in JSetL($\mathcal{FD}$) ha la forma generale

$$o_1.\texttt{neq}(o_2)$$

dove o_1, o_2 possono essere: espressioni aritmetiche oppure oggetti logici (variabile logica, insieme o lista). Il vincolo $o_1.\texttt{neq}(o_2)$ fallisce solo se il valore di o_1 è uguale al valore di 0_2. Se o_1 ed o_2 sono due tipi di espressioni diverse, ad esempio o_1 è di tipo intero e o_2 è un insieme logico, allora il vincolo è sempre consistente. Le regole di riduzione di dominio possono essere applicate solo se o_1 e o_2 sono entrambe di tipo intero.

Es.1 Se ad esempio una variabile con dominio è posta diversa da una costante di tipo reale allora il vincolo viene risolto con successo senza nessuna riduzione di dominio. Allo stesso modo se la costante è intera ma non è un valore compreso nel dominio della variabile. Ad esempio:

```
//aggiunge i vincoli di dominio x in [1..4], y in [0..2]
Solver.add(x.dom(1, 4));
Solver.add(y.dom(0, 2));


//aggiunge i vincoli x diverso da 2.2, y diverso da 8
Solver.add(x.neq(2.2));
Solver.add(y.neq(8));
```

```
Solver.solve();
```

L'output prodotto sarà

```
Lvar: x, id: 0, is NOT initialized., domain:Int[1..4]
Lvar: y, id: 0, is NOT initialized., domain:Int[0..2]
```

Es.2 Se la variabile x è posta diversa dalla costante intera k e questa costante coincide con un estremo del dominio di x allora (si veda il paragrafo 3.2.5) il valore k viene tolto da D_x.

```
//aggiunge il vincolo di dominio x in [1..4]
Solver.add(x.dom(1, 4));

//aggiunge il vincolo x diverso da 1
Solver.add(x.neq(1));

Solver.solve();
```

L'output prodotto sarà

```
Lvar: x, id: 0, is NOT initialized., domain:Int[2..4]
```

Es.3 Se invece x è posta diversa dalla costante intera k e questa costante non coincide con un estremo del dominio di x allora (si veda il paragrafo 3.2.5) il problema viene spezzato in due problemi alternativi. Il primo in cui $x < k$ e il secondo in cui $x > k$.

```
//aggiunge i vincoli di dominio
Solver.add(x.dom(1, 4));
```

```
//aggiunge il vincolo x diverso da due
Solver.add(x.neq(2));

//ricerca della prima soluzione
Solver.solve();
System.out.print("Prima soluzione: ");
System.out.println(x);

//ricerca della seconda soluzione
Solver.nextSolution();
System.out.print("\nSeconda soluzione: ");
System.out.println(x);
```

L'output prodotto sarà:

```
Prima soluzione  :
Lvar: x, id: 0, val: 1

Seconda soluzione:
Lvar: x, id: 0, is NOT initialized., domain:Int[3..4]
```

Il vincolo di diverso, come i vincoli di disuguaglianza, può contenere espressioni aritmetiche qualsiasi. Ad esempio:

$$z + 6 \neq x \cdot (y - 2)$$

```
z.sum(6).neq(x.mul(y.sub(4)))
```

4.4.4 Precedenza nelle Operazioni

L'estensione JSet($\mathcal{FD}$) ha anche aggiunto la possibilità di indicare precedenza nelle operazioni. Ad esempio

```
x.sum(y.mul(4))
```

indica l'espressione $x + (y \cdot 4)$ in cui viene prima eseguita la moltiplicazione $y \cdot 4$ e il suo risultato viene sommato a x. Invece l'istruzione

```
x.sum(y).mul(4)
```

indica l'espressione $(x+y)\cdot 4$ dove prima viene eseguita l'operazione di somma e poi quella di moltiplicazione. In JSetL invece non è così ma solo la sintassi del secondo esempio è corretta. Non è quindi possibile indicare nessuna precedenza nelle operazioni ma le operazioni vengono sempre svolte nell'ordine da sinistra verso destra. Il ché è evidentemente scomodo e obbliga spesso l'utente all'utilizzo di variabili di appoggio. Se ad esempio si vuole scrivere l'espressione aritmetica

$$(x \cdot 2) + (y \cdot 3)$$

in JSetL, utilizzando la precedenza sulle operazioni indicata dalle parentesi, si è costretti ad aggiungere almeno una d'appoggio z e riformulare l'espressione come

`y.mul(3).sum(z)`	$y \cdot 3 + z$
`z.eq(x.mul(2))`	$z = x \cdot 2$

mentre invece con JSetL($\mathcal{FD}$) può essere scritto in modo più naturale

`x.mul(2).sum(y.mul(3))` $(x \cdot 2) + (y \cdot 3)$

4.5 Labeling

La procedura di labeling deve essere esplicitamente richiesta dall'utente. Si può richiedere il labeling su di una variabile logica o su un insieme logico contenente variabili logiche. Nel secondo caso la procedura del labeling verrà applicata singolarmente ad ognuna delle variabili contenute nell'insieme.

L'utente ha due modi per richiamare il labeling. Il primo prevede di richiamare il metodo `label()` del risolutore: dove `M` può essere un istanza di

variabile logica (tipo `Lvar`) o un'istanza di insieme logico (tipo `LSet`).
Vediamo i due casi:

```
Solver.label(M);
```

In questo caso il risolutore procede automaticamente con la ricerca della soluzione.
Alternativamente l'utente può imporre un vincolo `label` su una o più variabili allo stesso modo con cui vengono aggiunti gli altri vincoli al risolutore.

```
Solver.add(M.label());
```

In questo caso il processo di ricerca della soluzione deve essere esplicitamente richiesto con l'istruzione:

```
Solver.solve();
```

Vediamo un semplice esempio. Eseguiamo il programma seguente senza procedura di labeling.

```
Solver.add(y.dom(1, 3));

//aggiunge i vincoli:
Solver.add(x.dom(0, 4));  //x in [0..4]
Solver.add(y.dom(-2..2)); //y in [-2..2]
Solver.add(x.lt(y));      //x < y

Solver.solve();

System.out.println(x);
System.out.println(y);
```

l'output prodotto sarà:

```
Lvar: x, id: 0, is NOT initialized., domain:Int[0..1]
Lvar: y, id: 1, is NOT initialized., domain:Int[1..2]
```

Tale output non rappresenta però una soluzione del problema in quanto le variabili non hanno valore assegnato. Vediamo come determinare le soluzioni del problema con l'aggiunta della procedura di labeling. Aggiungiamo al programma di sopra il seguente codice

```
Solver.add(x.label());
Solver.add(y.label());
Solver.solve();

//richiede il labeling su x e y
System.out.println(x);
System.out.println(y);

//ricerca di altre soluzioni
while(Solver.nextSolution()){
  System.out.println("--- ");
  System.out.println(x);
  System.out.println(y);
}
```

in questo caso l'output prodotto rappresenterà le tre possibili soluzioni del problema:

```
Lvar: x, id: 0, val: 0
Lvar: y, id: 1, val: 1
---
Lvar: x, id: 0, val: 0
Lvar: y, id: 1, val: 2
---
Lvar: x, id: 0, val: 1
Lvar: y, id: 1, val: 2
```

4.6 Esempio di un programma

Per dare una visione di insieme dell'utilizzo della libreria JSetL($\mathcal{FD}$) vediamo un intero programma che risolve il problema $SEND + MORE = MONEY$ introdotto nel paragrafo 2.1.3. Una più accurata descrizione del codice verrà data nel paragrafo 7.1.1.

```
//1.Dichiarazione delle variabili del problema
Lvar ls = new Lvar("S");
Lvar le = new Lvar("E");
Lvar ln = new Lvar("N");
Lvar ld = new Lvar("D");
Lvar lm = new Lvar("M");
Lvar lo = new Lvar("O");
Lvar lr = new Lvar("R");
Lvar ly = new Lvar("Y");
Lvar send = new Lvar("send");
Lvar more = new Lvar("more");
```

```
Lvar money = new Lvar("money");

//2.Crea un insieme con le variabili del problema
Object[] lettere = {ls, le, ln, ld, lm, lo, lr, ly};
LSet lett = new ConcreteLSet(lettere);

//3.Dichiarazione del risolutore
SolverClass solver = new SolverClass();

//4.Ciclo per assegnare ad ogni variabile un dominio
for (int i = 0; i < 8; i++) {
  solver.add(((Lvar) lett.get(i)).dom(0, 9));
}

//5.Esclude dal dominio di 'S' e 'M' il valore 0
//S > 0
solver.add(ls.gt(0));
//M > 0
solver.add(lm.gt(0));

//6.allDifferent(S,E,N,D,M,O,R,Y)
solver.add(lett.allDistinct());

//7.Impone i vincoli aritmetici
//SEND = S*1000 + E*100 + N*10 + D
solver.add(send.eq(ls.mul(1000).sum(le.mul(100)
  .sum(ln.mul(10).sum(ld)))));
//MORE = M*1000 + O*100 + R*10 + E
solver.add(more.eq(lm.mul(1000).sum(lo.mul(100)
```

```
  .sum(lr.mul(10).sum(le)))));
//MONEY = M*10000 + O*1000 + N*100 + E*10 + Y
solver.add(money.eq(lm.mul(10000).sum(lo.mul(1000)
  .sum(ln.mul(100).sum(le.mul(10).sum(ly))))));
//MONEY = SEND + MORE
solver.add(money.eq(send.sum(more)));

//8.Richiede il labeling
solver.add(lett.label());

//9.Lancia il processo di ricerca della soluzione
solver.solve();
```

Capitolo 5

Realizzazione di JSetL($\mathcal{FD}$)

In questo capitolo vedremo quali sono state le principali scelte di progettazione per la realizzazione del risolutore a vincoli su domini finiti descritto nel capitolo precedente e per la sua integrazione con il risolutore su vincoli insiemistici, presente in JSetL. Prima però è necessario vedere i principali aspetti implementativi della libreria JSetL.

5.1 Trattamento di vincoli e variabili logiche in JSetL

In questo paragrafo vediamo come sono implementati i principali aspetti relativi alla gestione dei vincoli in JSetL. Per maggiori e più dettagliate informazioni si veda la Tesi di Elisabetta Poleo [15].

5.1.1 Il Risolutore di vincoli (`SolverClass`)

In JSetL il risolutore di vincoli è rappresentato dalla classe `SolverClass`. Questa classe contiene un oggetto `CS` di tipo `ConstraintStore` dove vengono memorizzati i vincoli del problema. Il risolutore passa ad uno ad uno i vincoli contenuti nel `CS` ad un'oggetto di tipo `RewritingCostraintsRules`.

Questo oggetto ha il compito di agire sul singolo vincolo per determinarne o meno la consistenza. Inoltre si occupa di spezzare vincoli non in forma risolta in vincoli più "semplici". Dalla sostituzione del vincolo con uno dei suoi figli si crea un problema "semplificato" da cui si può portare avanti la ricerca della soluzione. Se questo problema dovesse risultare inconsistente (o si volesse cercare un'altra possibile soluzione) si procede con la costruzione di un altro dei possibili problemi sostituendo un altro dei figli al vincolo originale.
Il meccanismo che permette di seguire le varie alternative dovute allo spezzamento di un vincolo è implementato da un oggetto di tipo `Backtracking`. Prima che un il problema venga sostituito con una sua versione "semplificata" viene salvato lo stato delle variabili e del `CS` in modo che, in caso di fallimento, sia possibile recuperare questa situazione per poter procedere con una strada alternativa.

5.1.2 Legami tra variabili logiche

Le variabili logiche in JSetL sono implementate dalla classe `Lvar`.

Una variabile logica può avere un valore di qualsiasi tipo (variabile inizializzata) o non avere nessun valore (variabile non inizializzata). Per questo motivo nella classe `Lvar` è presente l'attributo `val` di tipo `Object` per rappresentare il valore della variabile logica e l'attributo booleano `iniz` per indicare se la variabile è inizializzata o no.

Un oggetto della classe `Lvar` può essere **legato** ad un altro oggetto di tipo `Lvar` tramite il campo `equ` di `Lvar`. Quando due o più variabili vengono vincolate ad essere uguali (ad esempio $x = y$) dapprima si verifica che il vincolo sia consistente e poi si legano queste variabili con una catena di puntatori di modo che fare riferimento ad una variabile di questo insieme piuttosto che ad un'altra sia indifferente. Questa soluzione di fatto realizza la nozione

di **unificazione** tra variabili logiche e permette una gestione efficiente della propagazione dell'uguaglianza. Vediamo alcuni esempi.

Es.1 Un problema descritto dalla serie di vincoli $x = 3, y = 4, z = x, z = y$, quando viene risolto il vincolo $z = x$, non viene assegnato il valore di x a z ma viene creato un riferimento (puntatore) dalla variabile z alla variabile x, in modo tale l'ultimo vincolo $z = y$ viene direttamente interpretato come $x = y$.

Es.2 Il problema inconsistente $x = y, x \neq y$, le due variabili sono non inizializzate, e quindi nel secondo vincolo non ci sono valori da confrontare per poterne determinare la diversità o meno, ma il risolutore è comunque in grado di determinare la non consistenza del problema perché il primo vincolo **unifica le due variabili** rendendole la stessa variabile.

Es.3 Allo stesso modo dell'esempio precedente fallirebbe il problema su insiemi $S = Q, S \neq \emptyset, S \cap Q = \emptyset$.

5.2 Implementazione dei domini in JSetL($\mathcal{FD}$)

In questo paragrafo viene fornita una descrizione di come i concetti visti nel capitolo 3 sono stati da noi implementati.

5.2.1 La classe `Interval`

Gli intervalli sono largamente usati nella libreria in quanto sono l'oggetto matematico utilizzato per rappresentare i domini delle variabili. Nella libreria JSetL esiste una classe chiamata `Int`. Nell'estensione JSet($\mathcal{FD}$) si è preferito cambiarne il nome della classe in `Interval` poiché il nome `Int` è stato ritenuto ambiguo per il suo utilizzo nei linguaggi di programmazione per indicare il tipo built-in intero. Inoltre nella classe `Interval` sono stati aggiunti alcuni nuovi metodi illustrati di seguito.

Il costruttore con un solo parametro intero `Integer(int a).` Questo metodo costruisce un intervallo dove l'estremo destro coincide con l'estremo sinistro uguale al valore del parametro `a` ovvero l'intervallo $[a..a]$. Questo costruttore si va ad aggiungere ai costruttori già esistenti `Integer(int a, int b)` che costruisce un intervallo $[a..b]$ e al costruttore senza parametri `Integer()` che costruisce l'intervallo che ha come estremo sinistro $-\infty$ e come estremo destro $+\infty$.

Il costruttore con un solo parametro non è stato introdotto per necessità (`Integer(a)` costruisce `Integer(a, a)`) ma per comodità. Come illustrato nel paragrafo 3.2, nell'applicare le regole di riduzione dei domini si generalizza una variabile inizializzata con valore intero ad una variabile con associato un dominio singoletto contenente quel valore. Allo stesso modo questa generalizzazione è stata fatta a livello di implementazione. Sono quindi frequenti i casi in cui è necessario costruire un intervallo singoletto.

Il metodo `Interval intersec(Interval intervallo).` Questo metodo ritorna il risultato dell'intersezione tra l'intervallo su cui è chiamato e l'intervallo che viene passato come parametro. Se l'intersezione è vuota il metodo ritorna il valore `null`. Questo metodo è molto utile, dato che la riduzione dei domini per mezzo di vincoli prevede sempre l'intersezione di intervalli.

Il metodo `boolean isSingleton().` Il metodo ritorna `true` se l'intervallo invocante contiene un solo elemento. Quando una variabile ha come dominio associato un singoletto, il valore contenuto nell'intervallo viene assegnato alla variabile.

5.2.2 La classe `Lvar`

L'attributo `domain.` In JSetL($\mathcal{FD}$) è stata necessaria l'aggiunta dell'attributo `Interval domain` nella classe `Lvar` per rappresentare il dominio as-

sociato alla variabile logica. Se questo parametro ha valore `null` si intende che nessun dominio è associato alla variabile.

Per leggere ed assegnare il dominio ad un oggetto di tipo `Lvar` vengono usati rispettivamente i metodi `getDomain()` e `setDomain()`. Questi due metodi sono leggermente più complicati dei comuni "getter & setter" che si utilizzano frequentemente nella programmazione a oggetti. I due metodi devono infatti confrontarsi con l'evenienza che la variabile logica faccia riferimento ad un'altra variabile logica, come meglio detto nel paragrafo 5.1.2. In questo caso l'operazione richiesta dovrà essere "inoltrata" alla variabile riferita.

I nuovi metodi della classe. I metodi introdotti da JSetL($\mathcal{FD}$) per la classe `Lvar`, altre ai metodi `getDomain()` e `setDomain()`, sono

```
Constraint dom(int a, int b)
Constraint label()
```

e servono entrambi a creare un vincolo sulla variabile logica su cui è invocato il metodo. Il loro significato quello è illustrato nei paragrafo 4.3 e 4.5 e precisamente:

- `x.dom(a,b)`
  associa a x il dominio $[a..b]$,
- `x.label()`
  richiede il labeling sulla variabile x.

5.2.3 Domini non limitati

Nella rappresentazione dei domini risulta utile poter avere domini senza limite superiore oppure inferiore. Si prenda ad esempio la variabile x senza nessun dominio associato. Fino ad ora il sistema non ha nessun elemento per dedurre

a che tipo appartiene la variabile x. Se però la variabile partecipa ad un vincolo definito su interi, come ad esempio $x > 2$, il sistema può dedurre che la variabile x è di tipo intero. Allora $x \in \mathbb{Z}$ e quindi $D_x = (-\infty .. + \infty)$. Procedendo poi con la riduzione di dominio per il vincolo $x > 2$ risulterà $D_x = [3.. + \infty)$.
In JSetL($\mathcal{FD}$) i valori speciali $-\infty$ e $+\infty$ sono rappresentati rispettivamente da `Integer.MIN_VALUE` e `Integer.MAX_VALUE`, il minimo e il massivo valore interi rappresentabili dal calcolatore.

Le operazioni di riduzione del dominio devono tenere conto dei valori speciali di cui sopra. Prendiamo ad esempio il problema $x > 3,\ y > 2,\ z = x \cdot y$ dove inizialmente nessuna variabile ha un dominio assegnato. I primi due vincoli stabiliscono $D_x = [4.. + \infty)$ e $D_y = [3.. + \infty)$. Nel considerare il terzo vincolo il sistema vede che z è coinvolta in un vincolo aritmetico e quindi considererà inizialmente $D_z = (-\infty .. + \infty)$ cioè z è un qualsiasi valore intero. In seguito alla regola di riduzione della moltiplicazione il nuovo dominio di z risulta essere $[(3 \cdot 2)..(+\infty \cdot +\infty))$. Essendo $+\infty$ rappresentato dal massimo valore intero del calcolatore, il calcolo dell'estremo destro di D_z produrrebbe un *overflow*.
In JSetL($\mathcal{FD}$) abbiamo isolato i casi speciali che non possono essere calcolati come operazioni su interi. In questo caso $D_z = [6.. + \infty)$.

5.3 La riduzione dei domini

Nell'estensione JSetL($\mathcal{FD}$) è stata introdotta una nuova classe chiamata `DomainHandler`. Al solver è stato poi aggiunta un'istanza di tale classe che ha il compito di applicare le regole di riduzione dei domini viste nel paragrafo 3.2.

L'instanza di `DomainHandler`, presente nel risolutore, integra il lavoro svolto dall'istanza della classe `RewritingConstranitsRules` contenuta nel

solver contribuendo alla risoluzione di quei vincoli che coinvolgono variabili che hanno esplicitamente o implicitamente associato un dominio intero. Ad esempio il vincolo di unificazione che involve le variabili x, y viene prima risolto dal `RewritingConstranitsRules` dopodiché il vincolo viene passato al `DomainHandler` dove se una o entrambe le variabili hanno un dominio associato vengono fatte le riduzioni del caso.

Come ultima operazione ogni regola di riduzione del dominio prevede una intersezione tra l'intervallo $\mathcal{FD}$-*consistent* e il dominio attuale della variabile. Questa operazione molto frequente è gestita nella classe `DomainHandler` dal metodo:

```
Interval updateDomain(Lvar var, Interval interv, Constraint s).
```

Il metodo `updateDomain()` prende tre parametri: una variabile logica `Lvar var` un intervallo `Interval interv` e un vincolo `Constraint s` e ritorna una variabile di tipo `Interval`. Il metodo si preoccupa di aggiornare il dominio della variabile logica con l'intersezione tra il dominio attuale della variabile `var.getDomain()` e `interv`. Se l'intersezione dovesse risultare vuota viene sollevata un'eccezione `Failure`. Se invece l'intersezione dovesse risultare un unico valore la variabile viene inizializzata con tale valore e il dominio della variabile viene messo a `null` ad indicare che non vi è nessun dominio associato.
Supponiamo ad esempio che durante il trattamento di un certo vincolo `s` imposto su `var` venga eseguito il seguente codice, e supponiamo che `var` sia non inizializzata

```
Interval d1 = new Interval(8,15);
Interval d2 = new Interval(0,10);
Interval d3 = new Interval(12,25);

updateDomain(var, d1, s);
updateDomain(var, d2, s);
updateDomain(var, d3, s);
```

Essendo la variabile `var` non inizializzata il sistema esegue la prima istruzione di `updateDomain` e (intersecando idealmente $\mathbb{Z}$ con [8..15]) assegna il dominio `d1` a `var`. Al termine della procedura si ha dunque `var` $\in$ [8..15]. Quando viene eseguita la seconda istruzione di `updateDomain` la variabile ha già un dominio e quindi il sistema esegue l'intersezione tra `var.getDomain()` (che a questo punto è l'intervallo [8..15]) e `d2` (l'intervallo [0..10]). Quindi il nuovo dominio di `var` sarà ([8..15] $\cap$ [0..10] =)[8..10]. L'esecuzione della terza istruzione solleva invece un un'eccezione `Failure` in quanto l'intersezione tra il dominio della variabile ([8..10]) e `d3` ([12..25]) è vuota.

5.4 Labeling

Come detto nel paragrafo 5.1.1 il risolutore JSetl attua la ricerca della soluzione spezzando vincoli non in forma risolta in vincoli più "semplici" e costruendo così l'albero di ricerca della soluzione. Questa tecnica viene riassunta in JSetL con il processo di *backtracking*. Il comportamento della procedura di labeling descritta nel paragrafo 3.4.1 funziona sostanzialmente allo stesso modo. Prende il dominio associato ad una variabile e lo spezza in due creando così da un unico problema due problemi alternativi: la prima alternativa vede la variabile con assegnato un valore del proprio dominio, diciamo k, la seconda alternativa vede il dominio della variabile privato del valore k. In modo tale si hanno due problemi più piccoli, la cui unione delle soluzioni

è equivalente all'insieme delle soluzioni del problema originario. Sfruttando questa analogia, JSetL($\mathcal{FD}$), integra il processo di labelinig nel processo di *backtracking*.

La richiesta di labeling su una singola variabile è trattata a tutti gli effetti come un vincolo sulla variabile. Tale vincolo viene quindi inserito nel `CS` e viene processato dal `RewritingConstraintsRules` che semplicemente ne delega la risoluzione al `DomainHandler` come capita per tutti i vincoli introdotti dalla gestione degli $\mathcal{FD}$. Il vincolo di labeling viene risolto però solo nel momento in cui la risoluzione e la propagazione dei vincoli è stata completata. Questo è importante perché è tramite la combinazione tra riduzione dei domini e propagazione dei vincoli che viene ridotto lo spazio di ricerca della soluzione ed è solo al termine di queste procedure che si è goduto a pieno dei vantaggi dovuti alle tecniche $\mathcal{FD}$.

Capitolo 6

Un'estensione a JSetL($\mathcal{FD}$): il vincolo globale `allDifferent`

Il vincolo *globale* `allDifferent` riveste particolare importanza nella programmazione logica a vincoli per il suo frequente utilizzo e la sua non così ovvia implementazione. Il problema si presenta naturalmente in una larga varietà di problemi che vanno dai puzzle, come il problema delle n regine, ai problemi di schedulazione e assegnamento. Per questi motivi è stato molto studiato in letteratura (si vedano [9, 10] per maggiori dettagli sul vincolo di `allDifferent`).

Perché il vincolo `allDifferent` sia consistente deve essere possibile assegnare, ad ognuna delle variabili coinvolte, un valore del proprio dominio tale che ogni variabile sia diversa dalle altre. Ad esempio il vincolo `allDifferent`(x_1, x_2, x_3) con $D_{x_1} = D_{x_2} = D_{x_3} = [1..2]$ è inconsistente. Non c'è modo infatti di assegnare valori di dominio alle variabili in modo che siano tutte diverse tra di loro.

Si hanno situazioni in cui valori di dominio, associati a variabili coinvolte nel vincolo `allDifferent`, non potranno mai essere assegnati altrimenti si determinerebbe sicuramente un'inconsistenza. Si consideri ad esempio il

CSP dato da `allDifferent`(x_1, x_2, x_3) con $D_{x_1} = [1..2], D_{x_2} = [1..2], D_{x_3} = [1..10]$. x_1 può assumere due possibili valori: 1 e 2. Se a x_1 viene assegnato il valore 1, a x_2 deve essere assegnato il valore 2, se invece a x_1 viene assegnato il valore 2, a x_2 deve essere assegnato il valore 1. In entrambi i casi a x_3 non potrà essere assegnato ne il valore 1 ne il valore 2. Fatta questa considerazione è possibile ridurre il dominio di x_3 a $D_{x_3} = [3..10]$.
La riduzione dei domini è molto importante nelle tecniche di risoluzione di problemi a vincoli perché, riducendo lo spazio della soluzione, aumentano la velocità di ricerca della stessa, d'altra parte però l'utilizzo di algoritmi molto raffinati può comportare un elevato costo computazionale e un conseguente calo di prestazioni.

6.1 Definizione ed uso del vincolo `allDifferent`

Vediamo per cominciare una definizione formale del vincolo `allDifferent`.

Definizione 6.1 (allDifferent) *Siano $x_1, x_2, \ldots, x_n$ variabili con i rispettivi domini $D_{x_1}, D_{x_2}, \ldots, D_{x_n}$. Allora*

`allDifferent`$(x_1, x_2, \ldots, x_n)$ *è consistente se*

$$(x_1, x_2, \ldots, x_n) \in \{(d_1, d_2, \ldots, d_n) \mid d_i \in D_{x_i}, d_i \neq d_j \text{ per } i \neq j\}.$$

Ad esempio se x, y, z sono tre variabili con dominio $D_x = D_y = D_z = [2..5]$ e si impone `allDifferent` (x, y, z), una possibile soluzione di tale problema è $x = 2,\ y = 3,\ z = 4$.

Vediamo come può essere modellato il noto problema delle *n regine* usando il vincolo `allDifferent`.

Esempio 6.1 *Si vuole modellare il problema delle n-regine descritto come: "Posizionare n regine su una scacchiera $n \times n$ $(n > 3)$ in modo che non si attacchino tra loro."*

Un modo di modellare questo problema è, come già visto nel paragrafo 2.1, di introdurre una variabile intera x_i $i = 1, 2, \ldots, n$ che rappresenta la regina posizionata sulla i-esima colonna, ognuna con dominio $[1..n]$ rappresentante le n possibili righe. Esprimiamo il "non attacco" con i vincoli come segue

$$\begin{aligned} x_i \neq x_j \qquad & (1 \leq i < j \leq n), \quad (1) \\ x_i - x_j \neq i - j \quad & (1 \leq i < j \leq n), \quad (2) \\ x_i - x_j \neq j - i \quad & (1 \leq i < j \leq n), \quad (3) \\ x_i \in \{1, 2, \ldots, n\} \quad & (1 \leq i \leq n). \qquad (4) \end{aligned}$$

I vincoli (1) stabiliscono che non è consentito che due regine si trovino sulla stessa riga mentre i vincoli (2) e (3) stabiliscono che non si trovino sulla stessa diagonale. Con una trasformazione algebrica su (2) e (3) possiamo riscrivere i vincoli nel modo seguente

$$\begin{aligned} &\texttt{allDifferent}(x_1, x_2, \ldots, x_n), \\ &\texttt{allDifferent}(x_1 - 1, x_2 - 2, \ldots, x_n - n), \\ &\texttt{allDifferent}(x_1 + 1, x_2 + 2, \ldots, x_n + n), \\ &x_i \in \{1, 2, \ldots, n\} \ (1 \leq i \leq n). \end{aligned}$$

6.2 Trattamento di `allDifferent`: decomposizione binaria

Una implementazione intuitiva del vincolo `allDifferent` su di n variabili è quella di creare per ogni coppia di variabili un vincolo di diverso. Questo procedimento è chiama decomposizione binaria del vincolo `allDifferent` .

La decomposizione binaria di $\texttt{allDifferent}(x_1, x_2, \ldots, x_n)$ è

$$\bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}.$$

Quindi usare il vincolo `allDifferent` diventa ne più ne meno che un'abbreviazione di scrivere i $\frac{1}{2}(n^2 - n)$ vincoli di diverso descritti dalla sua decomposizione binaria.

Questo tipo di implementazione presenta due svantaggi. Il primo è che comunque il risolutore dovrà riscrivere il vincolo globale come $\frac{1}{2}(n^2 - n)$ vincoli di diverso. Il secondo, più importante, è la perdita di informazione che si ha nel passare da un unico vincolo, dove la consistenza locale può essere considerata sul totale delle n variabili, a tanti vincoli binari dove la consistenza locale di ognuno può essere considerata solo sulla coppia isolata di variabili. Si ricordi infatti che il vincolo di diverso può operare una riduzione dei domini solo quando una delle due variabili coinvolte è un insieme singoletto. Vediamolo formalizzato nel seguente teorema.

Teorema 6.1 *Sia P il seguente CSP e P_{dec} lo stesso CSP in cui però il vincolo* `allDifferent` *è stato sostituito con la sua decomposizione binaria. Allora $\Phi_{HA}(P) \prec \Phi_A(P_{dec})$.*

Dimostrazione. Per mostrare che $\Phi_{HA}(P) \prec \Phi_A(P_{dec})$, si consideri il valore $d \in D_i$ per qualche i, rimosso per rendere P_{dec} arc consistent. Questa rimozione è stata fatta perché esiste un $x_j = d$ per qualche $j \neq i$. Ma allora $d \in D_i$ viene anche rimosso quando si rende P hyper-arc cosistent. Il contrario non è vero come illustrato nel controesempio.
Per qualche intero $n \geq 3$ consideriamo i CSP

$$P = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \textit{ per } i = 1, 2, \ldots, n-1, \\ x_n \in \{1, 2, \ldots, n\}, \\ \texttt{allDifferent}(x_1, x_2, \ldots, x_n). \end{cases}$$

$$P_{dec} = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \textit{ per } i = 1, 2, \ldots, n-1, \\ x_n \in \{1, 2, \ldots, n\}, \\ \bigcup_{1 \leq i < j \leq n} \{x_i \neq x_j\}. \end{cases}$$

Allora $\Phi_{HA}(P_{dec}) \equiv P_{dec}$, mentre

$$\Phi_{HA}(P) = \begin{cases} x_i \in \{1, 2, \ldots, n-1\} \text{ per } i = 1, 2, \ldots, n-1, \\ x_n \in \{n\}, \\ \texttt{allDifferent}(x_1, x_2, \ldots, x_n). \end{cases}$$

□

6.3 allDifferent come vincolo globale

Visti gli svantaggi che porta trattare il vincolo `allDifferent` come la sua decomposizione binaria, vedremo in questo paragrafo prima la teoria che sta alla base di un algoritmo `allDifferent` come vincolo globale e poi la sua implementazione.

Definizione 6.2 *Sia K l'insieme di variabili $\{x_1, x_2, \ldots, x_n\}$ e sia D_{x_i} il dominio associato alla variabile x_i per $i = 1, 2, \ldots, n$, definiamo*

$$D_K = \bigcup_{x_i \in K} D_{x_i}$$

6.3.1 Il Teorema di Hall

Un utile teorema per studiare l'implementazione di un algoritmo di riduzione dei domini per il vincolo `allDifferent` è il Teorema di Hall (Hall, 1935).

Teorema 6.2 *Se un gruppo di uomini e donne si sposa solo se sono stati precedentemente presentati a vicenda, allora un insieme completo di matrimoni è possibile se e solo se ogni sottoinsieme di uomini è stato collettivamente presentato ad almeno lo stesso numero di donne e viceversa*[1].

Vediamo la formulazione di questo teorema in termini del vincolo globale `allDifferent`.

[1] Assumiamo che i matrimoni siano ristretti a coppie di persone di sesso opposto

Teorema 6.3 (di Hall) *Il vincolo* `allDifferent`$(x_1, x_2, \ldots, x_n)$ *ha soluzione se e solo se*

$$|K| \leq |D_K|$$

per tutti i $K \subseteq \{x_1, x_2, \ldots, x_n\}$.

Dimostrazione. Chiamiamo **insieme K stretto** un insieme per cui valga $|K| = |D_K|$. Dimostriamo per induzione ipotizzando che il Teorema 6.3 sia valido per $k < n$.

Se esiste $d \in D_{x_n}$ tale che

$$x_1 \in D_{x_1} \backslash \{d\}, x_2 \in D_{x_2} \backslash \{d\}, \ldots, x_{n-1} \in D_{x_{n-1}} \backslash \{d\},$$
$$\texttt{allDifferent}(x_1, x_2, \ldots, x_{n-1})$$

ha soluzione allora il teorema è dimostrato. Quindi dobbiamo assumere il contrario e cioè esiste un sottoinsieme $K \subseteq \{x_1, x_2, \ldots, x_{n-1}\}$ con $|K| > |D_K \backslash \{d\}|$ per ogni $d \in D_{x_n}$.
Allora per induzione, `allDifferent`$(x_1, x_2, \ldots, x_{n-1})$, con $x_1 \in D_{x_1}, x_2 \in D_{x_2}, \ldots, x_{n-1} \in D_{x_{n-1}}$, ha una soluzione se e solo se per ogni $d \in D_{x_n}$ esiste un "insieme K stretto", $K \subseteq \{x_1, x_2, \ldots, x_{n-1}\}$ con $d \in D_K$. Scelto un qualsiasi "insieme K stretto". Senza perdita di generalità, $K = \{x_1, x2, \ldots, x_k\}$. Per induzione, `allDifferent`$(x_1, x_2, \ldots, x_k)$ ha soluzione, usando tutti i valori in D_K. Inoltre,

$$x_{k+1} \in D_{x_{k+1}} \backslash \{D_K\}, x_{k+2} \in D_{x_{k+2}} \backslash \{D_K\}, \ldots, x_n \in D_{x_n} \backslash \{d\},$$
$$\texttt{allDifferent}(x_{k+1}, x_{k+2}, \ldots, x_n)$$

ha soluzione.
Proseguendo per induzione, dal momento che ogni $L \subseteq \{x_{k+1}, x_{k+2}, \ldots, x_n\}$,

$$\left|\bigcup_{x_i \in L}(D_{x_i} \backslash D_K)\right| = \left|\bigcup_{x_i \in K \cup L} D_{x_i}\right| - |D_K|$$
$$\leq \text{(per induzione)}$$
$$|K \cup L| - |D_K| = |K| + |L| - |D_K| = |L|.$$

Allora `allDifferent`$\{x_1, x_2, \ldots, x_n\}$ ha soluzione, usando tutti i valori $D_K \cup D_L$. □

Esempio 6.2 *Si consideri il CSP seguente*

$$x_1 \in \{2,3\}, x_2 \in \{2,3\}, x_3 \in \{1,2,3\}, x_4 \in \{1,2,3\},$$
$$\texttt{allDifferent}(x_1, x_2, x_3, x_4).$$

Per ogni $K \subseteq \{x_1, x_2, x_3, x_4\}$ con $|K| \leq 3, K \leq |D_K|$.
Per $K = \{x_1, x_2, x_3, x_4\}$ in ogni caso $|K| > |D_K|$ e per il teorema 6.3 questo CSP non ha soluzione.

6.3.2 Trattamento del vincolo globale `allDifferent`

Un algoritmo in grado di garantire range consistency per il vincolo globale di `allDifferent` è stato introdotto da (Laconte, 1996). L'idea è quella di usare il Teorema di Hall per costruire l'algoritmo. Di seguito il teorema che mette in relazione il Teorema di Hall con la nozione di range consistency.

Teorema 6.4 *Definiamo l'intervallo* $I_K := [\min(D_K)..\max(D_K)]$. *Il vincolo* `allDifferent`$(x_1, x_2, \ldots, x_n)$ *è range consistent se e solo se nessun dominio* D_i *è vuoto e per ogni sottoinsieme* $K \subseteq \{x_1, x_2, \ldots, x_n\}$ *tale che* $|I_K| = |K|$ *implica* $D_i \cap I_K = \emptyset$ *per ogni* $x_i \notin K$.

In altre parole, se il minimo intervallo contenente l'unione dei domini di un sottoinsieme delle variabili vincolate è uguale alla cardinalità di questo insieme, allora nessun valore in questo intervallo deve essere contenuto nel dominio delle variabili non in K.

Definizione 6.3 *Un sottoinsieme K per cui valga la condizione descritta nel teorema 6.4, è chiamato insieme di Hall.*

Passiamo ora alla descrizione dell'algoritmo. L'idea di base è quella di trovare gli insiemi di Hall e per ognuno di questi insiemi K, rimuovere i valori in D_K dal dominio delle variabili non in K.

Un algoritmo naive potrebbe funzionare come segue. Per h che varia sopra i valori di massimo dominio e l che varia sui valori di minimo dominio, viene verificato quante variabili hanno il loro dominio contenuto in $[l..h]$. Se le variabili di questo tipo sono più di $|l..h| = (h - l) + 1$ allora il vincolo è inconsistente in quanto siamo nella situazione $|K| > |D_K|$. Se invece il numero di queste variabili è uguale all'ampiezza dell'intervallo $[l..h]$ allora siamo nella situazione $|K| = |D_K|$ e quindi l'intervallo deve essere tolto da tutte le altre variabili non in K.

```
ordina decrescente {x_1, x_2, ..., x_n} secondo max(D_{x_i})
ordina decrescente {x_1, x_2, ..., x_n} secondo min(D_{x_j})
for i ∈ {1, 2, ..., n}
    K := ∅
    h := max(D_{x_i})
    for j ∈ {1, 2, ..., n}
        l := min(D_{x_j})
        if [l..h] contiene D_{x_j}
            K := K ∪ {x_j}
        end if
        if |K| = |[l..h]|
            break
        end if
    for j ∈ {1, 2, ..., n}
        if x_j ∉ K
            rimuovi [l..h] da D_{x_j}
        end if
    end for
end for
```

Nel peggiore dei casi il costo dell'algoritmo è $O(n^3)$ dovuto ai tre cicli annidati. Uno per far variare h, uno per l e uno per x_i, ognuno di n passi.

La complessità dell'algoritmo può essere portata a $O(n^2)$ se si trova il modo di restringere i domini in $O(1)$. A questo scopo prima ordiniamo e salviamo i domini con due diversi ordinamenti. Il primo in ordine decrescente sul loro valore massimo e il secondo in modo decrescente sul loro valore minimo. Gli ordinamenti possono essere fatti in $O(n\log n)$ passi che è meno di $O(n^2)$.

A questo punto un ciclo esterno fa variare h su tutti gli estremi superiore dei domini in ordine decrescente e un ciclo interno fa variare l su tutti gli estremi inferiori dei domini in ordine decrescente. Durante il ciclo interno si tiene conto di quali variabili non sono contenute, ma intersecano, l'intervallo $[l..h]$. Se si individua un insieme di Hall il ciclo interno procede fino alla fine togliendo l'intervallo $[l..h]$ da tutte le variabili non ancora verificate. Questa rimozione non aumenta la complessità dell'algoritmo. Dopodiché si procede a togliere l'intervallo $[l..h]$ dalle variabili che, si era precedentemente verificato, intersecano $[l..h]$.
Quest'ultima operazione ha un costo $O(n)$, quindi sommato al costo del ciclo interno otteniamo $O(2 \cdot n) = O(n)$. Considerando il ciclo esterno, il costo dell'intero algoritmo risulta essere $O(n^2)$.

6.4 Inserimento di `allDifferent` in JSetL($\mathcal{FD}$)

In JSetL il vincolo `allDifferent` è presente ed è definito su insiemi logici. Se S è l'insieme $\{x_1, x_2, \ldots, x_n\}$, dove $x_1, x_2, \ldots, x_n$ sono variabili logiche non inizializzate, il vincolo S.`allDifferent`$(x_1, x_2, \ldots, x_n)$ impone che le variabili $x_1, x_2, \ldots, x_n$ siano tutte diverse tra loro.

L'implementazione adottata in JSetL è la decomposizione binaria. Visti i limiti di questa implementazione su insiemi di variabili logiche con dominio si è deciso di introdurre nell'estensione JSetL($\mathcal{FD}$) un nuovo vincolo che implementi `allDifferent` come vincolo globale seguendo l'idea di (Laconte,

1996). Questo nuovo vincolo è stato chiamato `allDistinct`. Si è optato per la convivenza dei due vincoli perché così facendo si hanno a disposizione le due implementazioni.
Da notare che il vincolo `allDistinct` non porta nessun vantaggio se le variabili coinvolte non sono esplicitamente di tipo intero (non hanno un dominio associato).

Nell'implementazione fatta di `allDistinct` si è dovuto tenere conto della rappresentazione dei domini usata in JSetL($\mathcal{FD}$) (vedi paragrafo 3.1). L'algoritmo infatti prevede di rimuovere intervalli dai domini delle variabili. Così facendo però si può incorrere nell'eventualità di dovere spezzare il dominio di una variabile. Questo porterebbe ad avere il dominio di una variabile non più come un unico intervallo ma come l'unione di due intervalli disgiunti.

Si prenda per esempio il dominio $D_x = [1..10]$ associato a x. Se durante il trattamento del vincolo `allDistinct` si rilevasse un insieme di Hall $[3..5]$ e x fosse una delle variabili vincolate, allora il dominio di x verrebbe ridotto a $D_x = ([1..2] \cup [6..10])$, tale rappresentazione del dominio non è supportata dal sistema. D'altra parte se si decidesse di di rappresentare D_x come $[\min(D_x)..\max(D_x)]$ si perderebbe tutta il *potere riduttivo* del vincolo. Per aggirare questo problema si è usata la tecnica già utilizzata per la situazione analoga a cui può portare il vincolo di diverso descritto nel paragrafo 3.2.5. Ogni qualvolta sia necessario spezzare il dominio di una variabile non si fa altro che spezzare il CSP in due CSP figli. Nel primo CSP la variabile vincolata avrà come dominio l'intervallo sinistro (nell'esempio $D_x = [1..2]$), mentre nel secondo CSP, la variabile avrà come dominio l'intervallo destro (nell'esempio $D_x = [6..10]$).

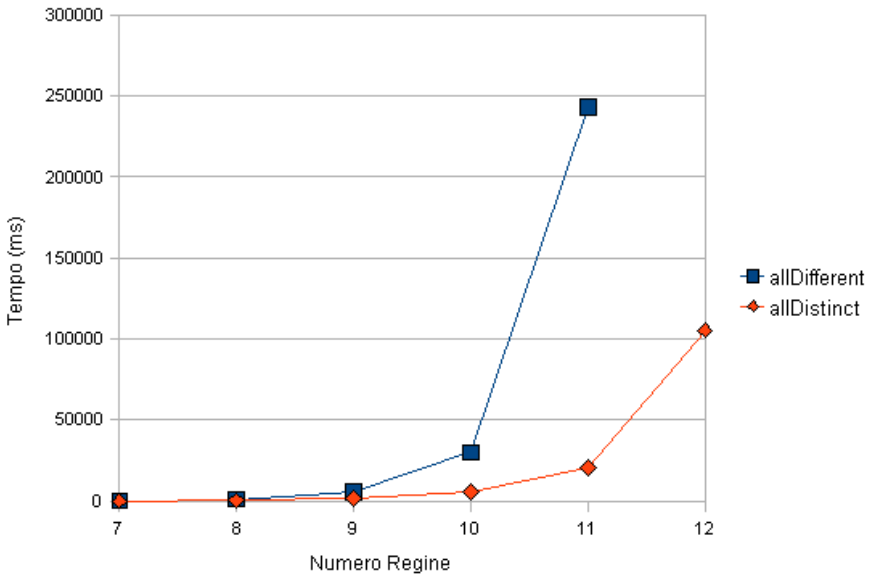
6.4.1 Test su `allDifferent` in JSetL($\mathcal{FD}$)

Per provare che effettivamente l'introduzione del vincolo `allDistinct` abbia apportato miglioramenti nei tempi di ricerca della soluzione si sono fat-

ti alcuni test alternando l'utilizzo del vincolo `allDifferent` che presenta l'implementazione con decomposizione binaria e il vincolo `allDistinct` implementato invece come vincolo globale. Vediamo di seguito quali risultati hanno conseguito i test eseguiti sul classico esempio delle n *regine* modellato come tre vincoli di `allDifferent` come descritto nell'esempio 6.1. Il codice Java usato per i test può essere visto a pagine 90.

Tempo per tutte le soluzioni (sec)

n	Soluzioni	`allDifferent`	`allDistinct`
7	40	0.25	0.187
8	92	1.109	0.442
9	352	5.703	1.64
10	724	30.375	5.687
11	2680	243.047	20.75
12	14200		105.391



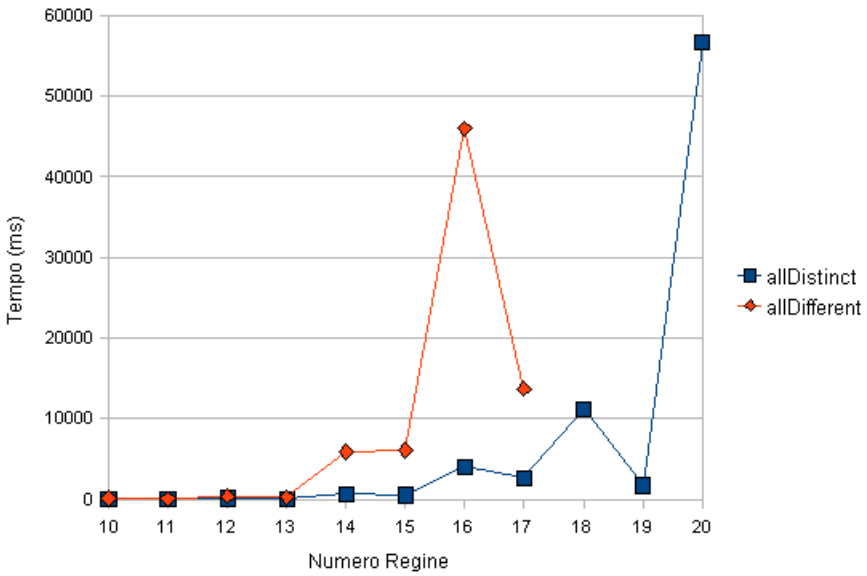
Dai grafici di pagina 83 e pagina 81 notiamo, come ci aspettavamo, che il tempo di ricerca di tutte le soluzioni cresce al crescere del numero delle

regine in entrambe le implementazioni. Infatti un numero più alto di regine significa allo stesso momento avere più variabili nel problema ma anche che i domini associati alle variabili sono più ampi. Ad esempio, per $n = 4$ si hanno quattro variabili con ognuna un dominio associato di quattro possibili valori il che significa un totale di $4^4 = 256$ possibili posizionamenti delle regine sulla scacchiera. Se invece $n = 5$, si hanno $5^5 = 3\ 125$ possibili posizionamenti. Grazie alle tecniche di riduzione dei domini il numero di possibili posizionamenti da "verificare" diviene molto inferiore ma è comunque evidente una crescita dell'albero di ricerca al crescere di n.

Osservando invece le curve date dai test sui tempi di ricerca di prima soluzione vediamo che non sempre al crescere di n cresce anche il tempo di ricerca (ad esempio il tempo di prima soluzione per $n = 17$ è inferiore al tempo per $n = 16$ in entrambe le implementazioni). Questo perché ricercare tutte le soluzioni di un problema significa costruire totalmente l'albero delle soluzioni (dove ogni foglia dell'albero è un fallimento o una soluzione) mentre la ricerca di prima soluzione interrompe il suo processo alla prima soluzione trovata. Se quindi la prima soluzione si trova vicina alla radice (in termini di costruzione dell'albero) il tempo de ricerca di tale soluzione sarà breva. Invece un tempo minore nella ricerca di tutte le soluzioni significa certamente un minore valore nel rapporto tra il tempo di risoluzione dei vincoli e la dimensione dell'albero di ricerca il ché non è necessariamente vero nel tempo di ricerca di prima soluzione. Sebbene il tempo di ricerca di prima soluzione sia molto importante all'atto pratico, in quanto spesso in problemi a vincoli si cerca *la soluzione* al problema e non si è interessati a quante e quali alternative si potrebbero avere, per quanto riguarda la misura delle prestazioni di un risolutore è più importante misurare il tempo di tutte le soluzioni, perché anche la diversa implementazione di un vincolo porta a una diversa costruzione dell'albero di ricerca.

Tempo di prima soluzione (sec)

n	allDifferent	allDistinct
10	0.156	0.109
11	0.109	0.94
12	0.407	0.156
13	0.297	0.109
14	5.875	0.703
15	6.125	0.563
16	45.937	4.031
17	13.718	2.688
18	$>$ 3 minuti	11.219
19	77.343	1.703
20	$>$ 3 minuti	56.594



6.5 Altre tecniche implementative del vincolo `allDifferent`

In JSetL($\mathcal{FD}$) per il trattamento del vincolo `allDifferent` è stato implementato un algoritmo sulla base dell'algoritmo indicato da (Laconte, 1996). Dai test eseguiti sul problema delle n Regine tale implementazione è risultata più efficiente della già esistente decomposizione binaria del vincolo. L'algoritmo di (Laconte, 1996) non è però l'unico conosciuto in letteratura (si vedano [9, 10]). Durante lo studio del problema del trattamento del vincolo `allDifferent` come vincolo globale, ci siamo soffermati in particolare sull'algoritmo indicato da (Régin, 1994). Tale algoritmo permette di determinare l'hyper-arc consistency del vincolo di `allDifferent`. Il funzionamento di tale algoritmo non verrà descritto in questo scritto in quanto necessita di nozioni di base relativi alla teoria dei grafi e le sue relative teorie dei flussi e dell'abbinamento massimo in grafi bipartiti [14]. Dopo lo studio dell'algoritmo di (Régin, 1994) si è deciso di non tentare una sua implementazione in JSetL($\mathcal{FD}$) poiché, nonostante la sua complessità ($O(m\sqrt{n})$) più bassa dell'algoritmo di (Laconte, 1996) ($O(n^2)$), necessita, a monte, la costruzione di strutture dati molto onerose in termine di costo computazionale. Per come è strutturato il risolutore di JSetL($\mathcal{FD}$) queste strutture andrebbero ricostruite numerose volte durante processo di ricerca della soluzione è il costo computazionale sarebbe sicuramente più alto di quanto non lo sia con l'algoritmi di (Laconte, 1996) che non prevede la costruzione di particolari strutture dati.

Capitolo 7

Esempi, Test e Confronti

Questo capitolo e suddiviso in due parti. Nelle prima vediamo due esempi di programmi completi che utilizzano la libreria JSetL($\mathcal{FD}$) per risolvere due classici problemi della programmazione a vincoli. Nella seconda vediamo invece a confronto i tempi di risposta per il problema delle n regine scritto con JSetL($\mathcal{FD}$) e con SWI Prolog [19]

7.1 Esempi

In questo paragrafo vediamo come scrivere programmi in linguaggio Java per alcuni problemi di programmazione logica a vincoli formulati utilizzando la libreria JSetL($\mathcal{FD}$). Ci soffermeremo più a lungo sul primo esempio per dare modo al lettore di prendere confidenza con le funzionalità della libreria.

7.1.1 $SEND + MORE = MONEY$

Vediamo il problema $SEND + MORE = MONEY$ descritto nel paragrafo 2.1.3, seguendo la modellazione

$$S, M \in \{1, 2, \ldots, 9\},\ E, N, D, O, R, Y \in \{0, 1 \ldots, 9\},$$
$$\texttt{allDifferent}(S, E, N, D, M, O, R, Y),$$

$$\begin{array}{rcccccccc} & 1000 \cdot S & + & 100 \cdot E & + & 10 \cdot N & + & D \\ + & 1000 \cdot M & + & 100 \cdot O & + & 10 \cdot R & + & E \\ = 1000 \cdot M & + & 1000 \cdot O & + & 100 \cdot N & + & 10 \cdot E & + & Y \end{array}$$

Di seguito vediamo il codice Java dove sono stati inseriti diversi commenti. Il numero che precede il commento serve ad identificare la sezione commentata. Di seguito al codice presentiamo una ampia spiegazione di ogni sezione.

```
import JSetL.*;

public class provaSendMoreMoney {

  public static void main(String[] args) throws Failure {

    //1.Dichiarazione delle variabili del problema
    Lvar ls = new Lvar("S");
    Lvar le = new Lvar("E");
    Lvar ln = new Lvar("N");
    Lvar ld = new Lvar("D");
    Lvar lm = new Lvar("M");
    Lvar lo = new Lvar("O");
    Lvar lr = new Lvar("R");
    Lvar ly = new Lvar("Y");
    Lvar send = new Lvar("send");
    Lvar more = new Lvar("more");
    Lvar money = new Lvar("money");

    //2.Crea un insieme logico contenente
    //  le variabili del problema
```

```
Object[] lettere = {ls, le, ln, ld, lm, lo, lr, ly};
LSet lett = new ConcreteLSet(lettere);

//3.Dichiarazione del risolutore
SolverClass solver = new SolverClass();

//4.Ciclo per assegnare ad ogni variabile
for (int i = 0; i < 8; i++) {
  solver.add(((Lvar) lett.get(i)).dom(0, 9));
}

//5.Esclude dal dominio di 'S' e 'M' il valore 0
//S > 0
solver.add(ls.gt(0));
//M > 0
solver.add(lm.gt(0));

//6.allDifferent(S,E,N,D,M,O,R,Y)
solver.add(lett.allDistinct());

//7.Impone i vincoli aritmetici
//SEND = S*1000 + E*100 + N*10 + D
solver.add(send.eq(ls.mul(1000).sum(le.mul(100)
  .sum(ln.mul(10).sum(ld)))));
//MORE = M*1000 + O*100 + R*10 + E
solver.add(more.eq(lm.mul(1000).sum(lo.mul(100)
  .sum(lr.mul(10).sum(le)))));
//MONEY = M*10000 + O*1000 + N*100 + E*10 + Y
solver.add(money.eq(lm.mul(10000).sum(lo.mul(1000)
```

```
      .sum(ln.mul(100).sum(le.mul(10).sum(ly))))));
    //MONEY = SEND + MORE
    solver.add(money.eq(send.sum(more)));

    //8.Richiede il labeling
    solver.add(lett.label());

    //9.Lancia il processo di ricerca della soluzione
    solver.solve();

    //10.Stampa il risultato
    System.out.print("  ");
    ls.print();le.print();ln.print();ld.print();
    System.out.print("  send +\n  ");
    lm.print();lo.print();lr.print();le.print();
    System.out.print("  more =\n ");
    lm.print();lo.print();ln.print();le.print();ly.print();
    System.out.print(" money ");
  }
}
```

Sezione 1. Nella sezione 1 vengono dichiarate le otto variabili logiche del problema S, E, N, D, M, O, R, Y, più tre variabili di appoggio *send*, *more* e *money* che serviranno per avere una formulazione più leggibile dei vincoli aritmetici della sezione 7. Per ogni variabile logica è possibile dichiarare un **nome esterno**. Ad esempio la variabile `ls` ha nome esterno `S`. Di seguito chiameremo le variabili logiche con il loro nome esterno.

Sezione 2. Il codice di questa sezione crea un vettore di variabili logiche chiamato `lettere` da cui costruisce un insieme logico `lett` contenete le variabili S, E, N, D, M, O, R, Y. Facciamo questo perché ci permette, nella

sezione 6, di applicare il vincolo `allDistinct` definito per insiemi logici e, nella sezione 8, di richiedere il labeling sulle 8 variabili con un unico vincolo.

Sezione 3. La sezione 3 non è altro che la dichiarazione del risolutore.

Sezione 4. Nella sezione 4 per ogni variabile dell'insieme `lett` aggiungiamo nel *CostraintStore* un vincolo di dominio. Questo vincolo indica che la variabile può assumere un valore intero compreso tra 0 e 9.

Sezione 5. In questa sezione vengono aggiunti due vincoli che indicano l'esclusione del valore 0 dai domini di S e M. Questa esclusione è stata indicata utilizzando i vincoli $S > 0$ e $M > 0$. Si noti che allo stesso scopo si possono utilizzare in alternativa i vincoli $S \neq 0$ e $M \neq 0$ utilizzando rispettivamente le istruzioni `ls.neq(0)` e `lm.neq(0)`.

Sezione 6. Questa riga di codice aggiunge il vincolo `allDistinct` sull'insieme di variabili `lett`. Il vincolo `allDistinct` non è altro che un'implementazione più efficiente del vincolo `allDifferent` come spiegato nel capitolo.

Sezione 7. Il codice della sezione 7 esprime il vincolo aritmetico descritto all'inizio del paragrafo. Tale vincolo è stato spezzato in tre parti per maggiore facilità di lettura del codice stesso.

Sezione 8. In questa sezione viene aggiunta nel constraint store del risolutore la richiesta di labeling sulle variabili nell'insieme `lett`.

Sezione 9. Arrivati a questo punto tutti i vincoli del problema e le richieste di labeling sono stati inseriti nel constraint store del risolutore. L'istruzione `solver.solve()` lancia il processo di ricerca della soluzione. Si noti che le due istruzioni di sezione 8 e sezione 9 possono essere sostituiti dall'unica istruzione `solver.label(lett)`.

Sezione 10. Questa sezione stampa la soluzione nel modo seguente.

```
 9567  send +
 1085  more =
10652  money
```

7.1.2 n Regine

Vediamo ora come scrivere un programma per il famoso problema delle n *regine*:

$$\texttt{allDifferent}(x_1, x_2, \ldots, x_n),$$
$$\texttt{allDifferent}(x_1 - 1, x_2 - 2, \ldots, x_n - n),$$
$$\texttt{allDifferent}(x_1 + 1, x_2 + 2, \ldots, x_n + n),$$
$$x_i \in \{1, 2, \ldots, n\}\ (1 \leq i \leq n).$$

utilizzando la libreria JSetL($\mathcal{FD}$). Di seguito il codice per la ricerca di tutte le soluzioni del problema per $n = 6$ ed alcune spiegazioni delle diverse sezioni del codice.

```
class Queens {

    //Dimensione del problema
    public static final int N =6;

    //1.Dichiarazione di 3 insiemi logici ognuno
    //  contenente N variabili logiche
    public static LSet pos = ConcreteLSet.mkset(N);
    public static LSet diag1 = ConcreteLSet.mkset(N);
    public static LSet diag2 = ConcreteLSet.mkset(N);

    public static void main (String[] args) throws Failure {

      SolverClass solver = new SolverClass();

        for(int i = 0; i < N; i++){
          //2.Assegna il dominio alla variabile Xi
```

```
    solver.add(((Lvar)pos.get(i)).dom(1, N));

    //3.Crea la variabile (Xi+i) e (Xi-i)
    solver.add(((Lvar)diag1.get(i))
      .eq(((Lvar)pos.get(i)).sum(i)));
    solver.add(((Lvar)diag2.get(i))
      .eq(((Lvar)pos.get(i)).sub(i)));
  }

  //5.Aggiunge i tre vincoli di allDifferent
  solver.add(pos.allDistinct());
  solver.add(diag1.allDistinct());
  solver.add(diag2.allDistinct());

  //6.Labeling e Ricerca della prima soluzione
  solver.label(pos);

  //7.Ricerca delle rimanenti soluzioni
  int i = 0;
  do{
    System.out.println();
    System.out.println("-------" + ++i + "------");
    soluzioneGrafica(pos);
  }while(solver.nextSolution());

  //8.Stampa il numero delle soluzioni trovate
  //  e il tempo di esecuzione
  System.out.print(N + " queens - " + i + " soluzioni"
  solver.printTime();
```

```
        return;

    }

//9.Una funzione per stampare graficamente la soluzione
private static void soluzioneGrafica(LSet pos2) {
  System.out.println();
  for(int i = 0; i <N; i++){
    for(int j = N-1; j>=0; j--)
      if((Integer) ((Lvar)pos2.get(j)).getValue() == i+1)
        System.out.print("Q ");
      else if((i+j)%2==0)
        System.out.print("* ");
      else
      System.out.print("° ");
      System.out.println();
  }
}
```

Sezione 1. In questa sezione ci sono le dichiarazione dei tre insiemi ognuno con n variabili logiche rappresentanti le regine. Questi tre insiemi sono utilizzati rispettivamente per indicare riga, diagonale SudOvest-NordEst e diagonale NordOvest-SudEst occupata da ognuna delle regine.

Sezione 2. La sezione 2 assegna ad ogni regina il suo dominio, e quindi le possibile n righe.

Sezione 3 e 4. Queste sezioni creano rispettivamente le variabili $x_i + i$ e $x_i - i$ per $(i = 1, 2, \ldots, n)$ che saranno utilizzati nella sezione 5. Per ogni regina, mettono in relazione la sua posizione di riga con le posizioni nelle relative diagonali.

Sezione 5. Nella sezione 5 vengono aggiunti nel constraint store del risolutore tre vincoli di `allDistinct` che non sono altro che i tre `allDifferent` visti nella descrizione del problema a inizio paragrafo.

Sezione 6. La sezione 6 incorpora con un unica istruzione due operazioni. La prima è la richiesta di labeling sulle variabili del problema. La seconda è la ricerca della prima soluzione.

Sezione 7. In questa sezione facciamo particolarmente attenzione all'ultima riga, il comando `solver.nextSolution()`. Questa istruzione genera un fallimento nel problema e lancia la ricerca di un'altra soluzione. Ritorna `true` se questa soluzione esiste. Quindi il ciclo della sezione 7 ci permette di trovare tutte le soluzioni.

Sezione 8. Nella sezione 8 notiamo il metodo `sover.printTime()` che stampa il tempo che è intercorso per la ricerca della soluzione.

Sezione 9. Questa sezione esegue una stampa della soluzione. La stampa è richiamata all'interno del ciclo di ricerca delle soluzioni in sezione 7. La sezione 8 e 9 forniscono l'output seguente.

```
-------1------

° * Q * ° *
* ° * ° * Q
° Q ° * ° *
* ° * ° Q °
Q * ° * ° *
* ° * Q * °

-------2------

° Q ° * ° *
* ° * Q * °
° * ° * ° Q
Q ° * ° * °
° * Q * ° *
* ° * ° Q °

-------3------

° * ° * Q *
* ° Q ° * °
Q * ° * ° *
* ° * ° * Q
° * ° Q ° *
* Q * ° * °

-------4------

° * ° Q ° *
Q ° * ° * °
° * ° * Q *
* Q * ° * °
° * ° * ° Q
* ° Q ° * °

6 queens - 4 soluzioni


time = 0.141 sec
```

7.2 Test e Confronti

In questo paragrafo mettiamo a confronto i tempi di risoluzione per il problema delle n regine ottenuti con JSetL($\mathcal{FD}$) con quelli ottenuti con SWI Prolog [19]. Si è scelto il confronto con SWI Prolog perché è un software freeware che fornisce un risolutore $\mathcal{FD}$. Il codice utilizzato per il programma JSetL($\mathcal{FD}$) è quello visto nel paragrafo 7.1.2 mentre il codice per il programma SWI Prolog riproposto qui sotto è quello fornito dalla dispensa di Agostino Dovier e Andrea Formisano dell'Università di Udine [18].

```
:- use_module(library(bounds)).

queens(N, Queens) :-
length(Queens, N), Queens in 1..N,
constrain(Queens),label(Queens).

constrain(Queens) :-
all_different(Queens),diagonal(Queens).

diagonal([]).

diagonal([Q|Queens]) :-
sicure(Q, 1, Queens),diagonal(Queens).

sicure(_A ,_B ,[]).

sicure(X,D,[Q|Queens]) :-
nonattacca(X,Q,D), D1 is D+1, sicure(X,D1,Queens).
```

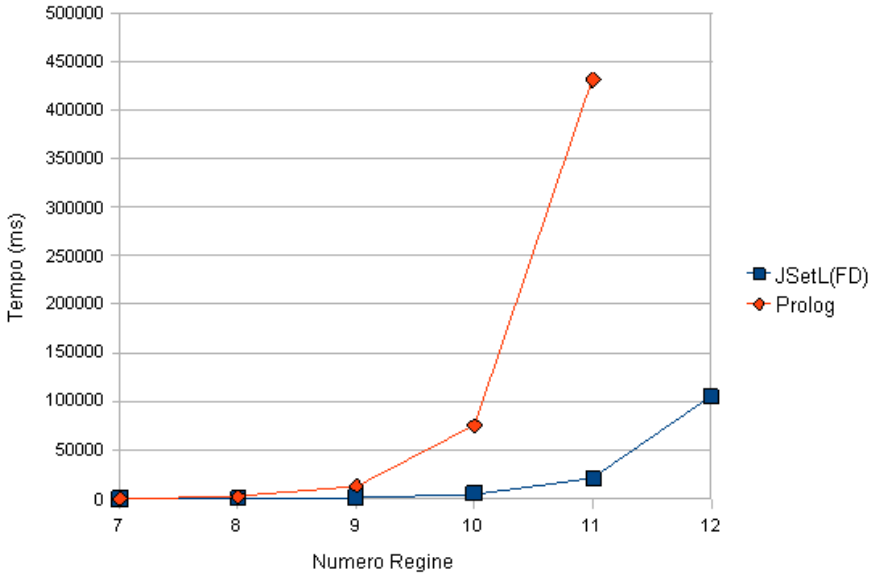
```
nonattacca(X,Y,D) :-
X + D #\= Y, Y + D #\= X.

allqueens(N,Queens):-
queens(N,Queens), fail.
```

Per richiedere la prima soluzione e il relativo tempo di ricerca per il problema con dieci regine si usa il predicato
`time(queens(10, [Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10])).`
Se invece si voule il tempo di ricerca per tutte le soluzioni si usa il predicato
`time(allqueens(10, [Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8,Q9,Q10])).`

Come già detto a proposito dei test fatti sul vincolo globale `allDifferent` nel paragrafo 6.4.1, sebbene i test sul tempo di prima soluzione siano importanti, perché all'atto pratico spesso è sufficiente un unica soluzione al problema, il confronto fra i tempi di ricerca di tutte le soluzione di un problema più indicativo quando si vogliono valutare le prestazioni di due diversi risolutore. Questo perché la prima soluzione può essere relativamente vicina o lontana dalla radice dell'albero di ricerca e questo dipende del problema, della sua modellazione e dall'implementazione del risolutore. Il processo per la ricerca di tutte le soluzioni, invece, deve comunque costruire l'intero albero di ricerca per assicurarsi che tutte le possibilità siano state esplorate.

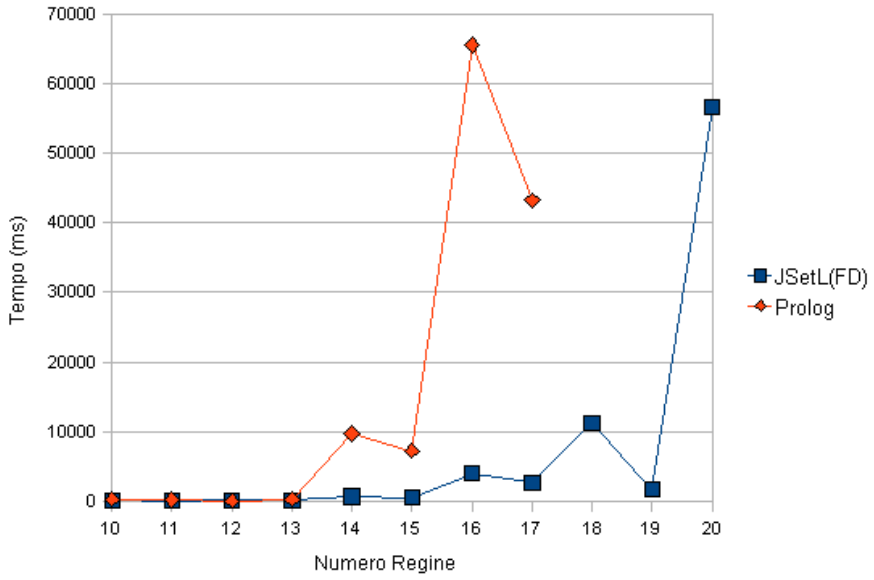
Tempo per tutte le soluzioni (sec)

n	Soluzioni	SWI Prolog	JSetL($\mathcal{FD}$)
7	40	0.510	0.187
8	92	2.750	0.442
9	352	13.810	1.64
10	724	75.520	5.687
11	2680	431.580	20.75
12	14200		105.391



Tempo di prima soluzione (sec)

n	SWI Prolog	JSetL($\mathcal{FD}$)
10	0.310	0.109
11	0.170	0.94
12	0.097	0.156
13	0.330	0.109
14	9.660	0.703
15	7.230	0.563
16	65.480	4.031
17	43.190	2.688
18	$>$ 3 minuti	11.219
19	26.300	1.703
20	$>$ 3 minuti	56.594



Come si vede dai grafici di pagina 98 e pagina 97 il programma scritto con la nostra libreria JSetL($\mathcal{FD}$) dà tempi di risposta più rapidi sia per la ricerca di prima soluzione sia per la ricerca di tutte le soluzioni. I risultati

possono essere dunque considerati soddisfacenti e indicano una buona qualità del lavoro svolto.

Capitolo 8

Conclusioni

In questo lavoro abbiamo affrontato il problema della progettazione e realizzazione di un risolutore per vincoli su domini finiti ($\mathcal{FD}$) e della sua integrazione con un risolutore per vincoli insiemistici all'interno della libreria JSetL [11].

Il risultato ottenuto è stato una nuova versione della libreria JSetL, denominata JSetL($\mathcal{FD}$), che offre un unico risolutore in grado di applicare le tecniche di risoluzione su vincoli insiemistici proprie di JSetL insieme a quelle dei risolutori $\mathcal{FD}$, in base al contesto del problema.

Questa integrazione è stata fatta in modo da essere trasparente all'utente finale. In altre parole sono stati preservati i vincoli e le funzionalità che erano già presenti in JSetL, ma è cambiato il modo in cui alcuni vincoli, principalmente quelli sugli interi, vengono trattati e risolti all'interno del risolutore fornito dalla libreria. In particolare, JSetL($\mathcal{FD}$) fornisce una nuova, più efficiente, implementazione del vincolo globale di `allDifferent`.

Un'ulteriore evoluzione della libreria potrebbe essere l'utilizzo di veri e propri insiemi di interi (non necessariamente intervalli) per la rappresentazione dei domini delle variabili. Ciò si potrebbe ottenere partendo dall'implementazione descritta in questa tesi ed aggiungendo la possibilità che i domini

delle variabili siano unione di più intervalli disgiunti. Questo permetterebbe una più naturale modellazione dei problemi $\mathcal{FD}$ e un diverso approccio per il trattamento dei vincoli 'di diverso' e `allDifferent` in cui risulta necessario spezzare il dominio delle variabili in unioni di intervalli disgiunti.

Bibliografia

[1] P. Van Hentenryck.
Constraint Satisfaction in Logic Programming.
The MIT Press, 1989.

[2] M. Dincbas, P. Van Hentenryck, H. Simons, A. Aggoun, T. Graf, F.Berthier
The Constraint Logic Programming Language CHIP.
ICOT, Tokyo, 1988.

[3] P. Codognet, D. Diaz
Compiling Constraints in clp(FD).
Journal of Logic Programming, 1996.

[4] K. R. Apt.
Principles of Constraint Programming.
Cambridge University Press, 2003.

[5] R. Dechter
Constraint Processing.
Elsevier Science & Technology Books , 2003.

[6] L. Console, E. Lamma, P. Mello, M. Milano
Programmazione logica e Prolog (II edizione)
UTET Libreria, 2006.

[7] CSPLib Home page
CSPLib: A problem library for constraints
`http://www.csplib.org/`

[8] Dovier, A., Piazza, C., Pontelli, E., and Rossi, G.
Sets and Constraint Logic Programming.
ACM Transactions on Programming Languages and Systems, 22(5):861–931, 2000.

[9] W. J. van Hoeve.
The Alldifferent Constraint: A Survey.
`http://www.math.unipd.it/~frossi/alldiff.pdf`

[10] B. Anastasatos.
Propagation Algorithms for the Alldifferent Constraint.
`http://ps.uni-sb.de/courses/seminar-ws04/papers/anastasatos.pdf`

[11] G. Rossi, E. Panegai, E. Poleo.
JSetL: a Java library for supporting declarative programming in Java.
Software Practice & Experience 2007; 37:115-149.

[12] A. Dal Palù, A. Dovier, E. Pontelli, G. Rossi.
A Constraint Logic Programming Framework for Effective Programming with Sets and Finite Domains.
Quaderno del Dipartimento di Matematica, n. 437, Universita' di Parma, March 2006.

[13] JSetL Home Page.
`http://prmat.math.unipr.it/~gianfr/JSetL/index.html`

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest.
Introduzione agli Algoritmi. Volume 2.
Jeckson Libri, 1994.

[15] E. Poleo.

JavaSet: una libreria Java per la

Programmazione con Vincoli (Insiemistici)

Tesi del Dipartimento di Matematica, a.a 2001-2002.

[16] D. Di Giorgio.

Gestione di insiemi ed operazioni insiemistiche in Java tramite l'integrazione tra la libreria JSetL e l'interfaccia Set di Java.

Tesi del Dipartimento di Matematica, a.a 2005-2006.

`http://www.cs.unipr.it/Informatica/Tesi/Delia_DiGiorgio_20070426.pdf`

[17] R. Amadini.

Defnizione e trattamento del vincolo di cardinalità insiemistica nella libreria JSetL

Tesi del Dipartimento di Matematica, a.a 2006-2007.

`http://www.cs.unipr.it/Informatica/Tesi/Roberto_Amadini_20071003.pdf`

[18] A. Dovier, A. Formisano.

Programmazione Dichiarativa con Prolog, CLP, CCP, e ASP

`http://users.dimi.uniud.it/~agostino.dovier/DID/lnc.pdf`

[19] SWI Prolog's Home Page.

`http://www.swi-prolog.org/`

[20] Java Platform, Standard Edition 6: API Specification.

`http://java.sun.com/javase/6/docs/api/`