UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI SCIENZE
MATEMATICHE, FISICHE e NATURALI

Corso di Laurea Specialistica in Informatica

Tesi di Laurea Specialistica

Studio e realizzazione in Java di domini e regole per la risoluzione di vincoli su interi e insiemi di interi

Relatore:
Prof. Gianfranco Rossi

Candidato:
Roberto Amadini

Anno Accademico 2010/2011

Ai miei genitori,
Bruno e Tiziana.

Indice

Capitolo 1

Introduzione

La programmazione dichiarativa (DP, *Declarative Programming*) è un paradigma di programmazione nel quale il programmatore descrive *cosa* il programma deve fare, piuttosto che *come* lo deve fare.

I principali vantaggi di questo paradigma riguardano la facilità di sviluppo e comprensibilità del programma, la sinteticità, la riusabilità e il parallelismo implicito.

La **programmazione con vincoli** (CP, *Constraint Programming*) può vedersi come una forma di DP che permette di manipolare esplicitamente vincoli (cioè, relazioni su opportuni domini sia simbolici che numerici) che trova numerose applicazioni pratiche nei più svariati ambiti (quali ad esempio bio-informatica, ottimizzazione, analisi finanziaria etc...).

Più precisamente, la CP permette la risoluzione di **problemi di soddisfacimento di vincoli** (CSP, *Constraint Satisfaction Problems*).
Un CSP è caratterizzato in generale da tre componenti fondamentali:

- un insieme finito di **variabili**, cioè entità che possono assumere un determinato numero di valori
- un insieme finito di **domini**, ognuno dei quali rappresenta i possibili valori che ciascuna variabile può assumere.
- un insieme finito di **vincoli**, cioè relazioni che le variabili del problema devono soddisfare.

Nel corso degli anni sono stati studiati e realizzati vari **risolutori di vincoli** (*constraint solvers*).
Alcuni di essi sono stati sviluppati all'interno di linguaggi '*ad hoc*'; la maggior parte invece utilizzano un linguaggio 'ospite' quale ad esempio C++,

Java o Prolog.[22][23][26]
In particolare, se il linguaggio utilizzato per la CP è di tipo logico si parla di programmazione logica con vincoli (CLP, *Constraint Logic Programming*).

L'approccio che forse ha avuto maggior successo nella CP è quello dei cosiddetti vincoli su **domini finiti**, cioè vincoli definiti su *insiemi finiti* di elementi.

In questo lavoro di tesi ci si occuperà dunque dello studio e dell'implementazione di risolutori di vincoli su domini finiti.

La base di tale lavoro sarà la libreria **JSetL**[16][25], un package Java che offre funzionalità per il supporto alla DP simili a quelle generalmente utilizzate nella CLP: strutture dati logiche, unificazione, ricorsione, risoluzione di vincoli, non-determinismo.
In particolare, JSetL fornisce molte delle funzionalità offerte da CLP($\mathcal{SET}$)[6], un linguaggio nel quale il dominio dei vincoli è costituito da insiemi possibilmente eterogenei, annidati e parzialmente specificati.

La libreria JSetL verrà quindi estesa mediante la realizzazione di:

(i) un risolutore di vincoli su *interi*, che migliorerà la precedente versione descritta in [14]

(ii) un risolutore di vincoli su *insiemi finiti di interi*, che seguirà l'approccio definito in [3].

Questi risolutori si basano sulle nozioni classiche di regole di *riduzione* dei domini, *propagazione* dei vincoli e *ricerca* della soluzione.

L'estensione di JSetL a **(i)** verrà chiamata JSetL($\mathcal{FD}$), mentre la corrispondente estensione a **(ii)** sarà denominata JSetL($\mathcal{FDS}$).

Il lavoro di tesi è organizzato nel seguente modo:

Nel Capitolo 2 verranno formalizzate la definizione di CSP, le proprietà di cui possono godere e le relazioni che possono intercorrere tra CSP.
Quindi, verrà descritto in termini astratti l'approccio generale di risoluzione di un CSP, soffermandosi in particolare sulle nozioni di *consistenza locale* associate a un certo vincolo.

Nel Capitolo 3 verranno definiti e discussi formalmente due domini finiti, gli *intervalli* e i *multi-intervalli* di interi, utilizzati in seguito per modellare il dominio delle variabili in CSP contenenti vincoli su interi.

In particolare, l'introduzione dei multi-intervalli rappresenterà una delle principali differenze rispetto alla precedente implementazione di JSetL($\mathcal{FD}$).[14]

Nel Capitolo 4 si parlerà di risoluzione di vincoli su interi.
Seguendo un approccio simile alla definizione dei linguaggi del prim'ordine nella logica classica, verranno introdotti il linguaggio $\mathcal{L}_{\mathcal{FD}}$ ed una sua interpretazione semantica Δ su $\mathbb{Z}$.
Ciò permetterà di definire formalmente tecniche di risoluzione per CSP basati su tale linguaggio, attraverso apposite regole di *riduzione* dei domini, *propagazione* dei vincoli e *ricerca* della soluzione.

Nel Capitolo 5, dopo una breve introduzione alla sopracitata libreria JSetL, verrà descritta la nuova versione di JSetL($\mathcal{FD}$), illustrando le principali caratteristiche delle nuove classi introdotte nel package: `Interval`, `MultiInterval`, `FDVar`.
Infine, verranno riportati alcuni programmi che utilizzano tali funzionalità per risolvere CSP su interi (ad esempio, il 'noto' problema cripto-aritmetico $SEND + MORE = MONEY$).

A questo punto comincia la seconda parte del lavoro di tesi, di fatto simmetrica alla prima, in cui verranno trattati problemi di soddisfacimento di vincoli su insiemi di interi.

Nel Capitolo 6 verrà discusso in modo formale il dominio degli *intervalli di insiemi di interi*, o più brevemente *set-interval*, che sarà utilizzato in seguito per modellare il dominio delle variabili in CSP contenenti vincoli su tali insiemi.
Ciò rappresenterà una novità assoluta in JSetL: in precedenza, la risoluzione di vincoli insiemistici era unicamente basata sul linguaggio CLP($\mathcal{SET}$).

Nel Capitolo 7, in modo speculare a quanto fatto nel capitolo 4, si parlerà di risoluzione di vincoli su insiemi di interi.
Verranno dunque introdotti il linguaggio $\mathcal{L}_{\mathcal{FS}}$, la sua estensione $\mathcal{L}_{\mathcal{FDS}}$ e le rispettive interpretazioni Σ e Φ.
Quindi, seguendo un approccio simmetrico rispetto a quanto fatto nel capitolo 4, verranno definite formalmente tecniche di risoluzione di CSP basati sul linguaggio $\mathcal{L}_{\mathcal{FDS}}$.

Nel Capitolo 8 verrà descritta l'estensione JSetL($\mathcal{FDS}$), illustrando le principali caratteristiche delle nuovi classi introdotte nella libreria:
`SetInterval` e `FSVar`.

Infine verranno riportati alcuni programmi che utilizzano tali funzionalità per risolvere CSP su insiemi di interi (ad esempio, il problema delle *Triple di Steiner*).

Nel Capitolo 9 verranno illustrati e discussi due programmi che utilizzano rispettivamente le estensioni JSetL($\mathcal{FD}$) e JSetL($\mathcal{FDS}$) per risolvere due particolari CSP:

- il problema delle delle *n-regine* (contenente esclusivamente vincoli su interi)
- il problema dei *social golfers* (contenente vincoli su insiemi di interi).

Oltre ai dettagli implementativi verranno illustrate e analizzate le prestazioni dei due programmi al variare della dimensione dell'input.

Nel Capitolo 10 infine ci sarà spazio per conclusioni ed eventuali lavori futuri.

Capitolo 2

Programmazione a vincoli

La programmazione a vincoli (CP, *Constraint Programming*) è un paradigma di programmazione nel quale le relazioni fra variabili sono espresse in forma di *vincoli* che devono essere soddisfatti.
Tali vincoli possono essere di vario genere, a seconda del *dominio* delle variabili: ad esempio vincoli di somma del tipo $Z = X + Y$ con X, Y e Z variabili di tipo intero oppure vincoli di inclusione del tipo $A \subseteq B$ con A e B insiemi.

Tra i vari paradigmi della CP, uno di quelli che ha avuto maggiore successo è basato sui *domini finiti*: i vincoli sono definiti su insiemi finiti di elementi. Attraverso la CP su domini finiti, è possibile risolvere in modo elegante ed efficiente problemi 'noti' in letteratura come Map Coloring, SEND+MORE=MONEY, n-Queens, eccetera.
Una delle prime e più comuni forme di CP è la programmazione logica a vincoli (CLP, *Constraint Logic Programming*), che di fatto consiste nell'integrazione di vincoli all'interno di un linguaggio logico.

Attualmente, la maggior parte delle implementazioni basate su Prolog includono una o più librerie per supportare la CLP.
Tuttavia, la programmazione a vincoli è possibile anche in linguaggi imperativi includendo apposite librerie.
In questo capitolo, dopo aver definito formalmente cos'è un problema di soddisfacimento di vincoli, ci si soffermerà in particolare su alcune nozioni di *consistenza locale* associate a un vincolo.

2.1 Problemi di soddisfacimento di vincoli

Definiamo ora formalmente cosa si intende per problema di soddisfacimento di vincoli e come vengono trattati tali problemi.

Definizione 2.1 (*CSP*)**.** Un problema di soddisfacimento di vincoli (**CSP**, *Constraint Satisfaction Problem*) è una tripla $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ dove:

- $\mathcal{V} = \{x_1, \ldots, x_n\}$ è un insieme *finito* di **variabili**
- $\mathcal{D} = D_1 \times \ldots \times D_n$ è una *n-upla* di **domini** tali che $D_i = dom(x_i)$ per ogni $i = 1, \ldots, n$
- $\mathcal{C} = \{c_1, \ldots, c_m\}$ è un insieme *finito* di **vincoli di arità** $0 \leq k \leq n$ definiti su sottoinsiemi di $\mathcal{V}$.
  Formalmente, per ogni $c \in \mathcal{C}$, se c ha arità k allora esiste un insieme di indici $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ tale che $c \subseteq D_{i_1} \times \ldots \times D_{i_k}$

Definizione 2.2 (*Soddisfacimento*)**.** Sia $d = \langle d_1, \ldots, d_n \rangle \in \mathcal{D}$ una n-upla di valori dei domini. Si dice che d **soddisfa** un vincolo $c \in \mathcal{C}$ (definito sulle variabili $x_{i_1}, \ldots, x_{i_k}$) se e soltanto se $\langle d_{i_1}, \ldots, d_{i_k} \rangle \in c$.

Se dato un $c \in \mathcal{C}$ esiste *almeno* una $d \in \mathcal{D}$ che lo soddisfa, allora c si dice **soddisfacibile**. Se viceversa non esiste una tale d, il vincolo c è **insoddisfacibile**.

Se $d \in \mathcal{D}$ soddisfa *ogni* $c \in \mathcal{C}$ allora d è una **soluzione** di $\mathcal{P}$.

Definizione 2.3 (*Consistenza di un CSP*)**.** Un CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ si dice:

- **consistente**, se ammette almeno una soluzione $d \in \mathcal{D}$
- **inconsistente**, se non ammette alcuna soluzione
- **risolto**, se per ogni $i = 1, \ldots, n$ si ha che $D_i = \{d_i\}$ e $\langle d_1, \ldots, d_n \rangle$ è una soluzione di $\mathcal{P}$
- **fallito**, se esiste $j \in \{1, \ldots, n\}$ tale che $D_j = \varnothing$ oppure per ogni $i = 1, \ldots, n$ si ha che $D_i = \{d_i\}$ ma $\langle d_1, \ldots, d_n \rangle$ non è soluzione di $\mathcal{P}$.

Osservazione 1 (Incompletezza). Se un CSP $\mathcal{P}$ è fallito, allora è sicuramente inconsistente (non trovo alcuna soluzione). Inoltre, se $\mathcal{P}$ è risolto è sicuramente consistente (esiste una soluzione).

Tuttavia, se $\mathcal{P}$ non è fallito potrebbe comunque essere inconsistente.
Si consideri infatti il CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ dove:

- $\mathcal{V} = \{x_1, x_2, x_3\}$
- $\mathcal{D} = D_1 \times D_2 \times D_3$ con $D_1 = D_2 = D_3 = \{0, 1\}$
- $\mathcal{C} = \{c_1, c_2, c_3\}$ con $c_1 \equiv x_1 \neq x_2$, $c_2 \equiv x_2 \neq x_3$ e $c_3 \equiv x_3 \neq x_1$

Si può notare come $\mathcal{P}$ sia non risolto, non fallito ma comunque inconsistente.

Definizione 2.4 (*Relazioni fra CSP*)**.** Dato un CSP $\mathcal{P}$, posso definire l'**insieme delle soluzioni** di $\mathcal{P}$ come

$$\xi(\mathcal{P}) \stackrel{def}{=} \{d \in \mathcal{D} : d \text{ è soluzione di } \mathcal{P}\}$$

Siano ora $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ e $\mathcal{P}' = \langle \mathcal{V}', \mathcal{D}', \mathcal{C}' \rangle$ due CSP. Si dice che:

- $\mathcal{P}$ è **equivalente** a $\mathcal{P}'$, e si scrive $\mathcal{P} \equiv \mathcal{P}'$, se e soltanto se $\xi(\mathcal{P}) = \xi(\mathcal{P}')$
- $\mathcal{P}$ è **minore o uguale** a $\mathcal{P}'$, e si scrive $\mathcal{P} \preceq \mathcal{P}'$, se e soltanto se per ogni $i = 1, \ldots, n$ si ha che $D_i \subseteq D'_i$
- $\mathcal{P}$ è **(strettamente) minore** di $\mathcal{P}$, e si scrive $\mathcal{P} \prec \mathcal{P}'$, se e soltanto se $\mathcal{P} \preceq \mathcal{P}'$ ed esiste $j \in \{1, \ldots, n\}$ tale che $D_j \subset D'_j$.

Definizione 2.5 (*Risolutore di vincoli*)**.** Un **risolutore di vincoli** (*Constraint Solver*) è costituito da un insieme di algoritmi e strutture dati che permettono, dato in input un CSP $\mathcal{P}$, di ottenere in output un CSP semplificato $\mathcal{P}'$ tale che $\mathcal{P}' \preceq \mathcal{P}$ e $\mathcal{P}' \equiv \mathcal{P}$.
Tale processo è detto di **risoluzione di vincoli** (*Constraint Solving*).
Un risolutore di vincoli si dice **corretto** se:

$$\mathcal{P}' \text{ fallito} \implies \mathcal{P} \text{ inconsistente} \quad \text{e} \quad \mathcal{P}' \text{ risolto} \implies \mathcal{P} \text{ consistente.}$$

Si dice invece **completo** se:

$$\mathcal{P}' \text{ inconsistente} \implies \mathcal{P} \text{ fallito} \quad \text{e} \quad \mathcal{P}' \text{ consistente} \implies \mathcal{P} \text{ risolto.}$$

Dato un CSP $\mathcal{P}$ l'obiettivo (*goal*) è quindi trovare una soluzione (o tutte le soluzioni) attraverso un fissato criterio di risoluzione.
Ci si può quindi chiedere quali tecniche adottare per risolvere un CSP, ovvero quale strategia deve implementare un risolutore per giungere (eventualmente) alla soluzione.
Senza dubbio, la correttezza è un requisito necessario. Ovviamente, anche la completezza è desiderabile; tuttavia, raggiungere questo obiettivo in modo efficiente (sopratutto nei casi reali) è spesso un'impresa ardua.

Ad esempio, utilizzare una tecnica 'completa' di tipo '*Generate & Test*' (genero e testo ogni possibile soluzione) risulterebbe nella stragrande maggioranza dei casi altamente inefficiente (nell'ordine di $\prod_{i=1}^{n} |D_i|$ nel caso peggiore).

Per ridurre il costo computazionale la programmazione a vincoli utilizza un processo chiamato **propagazione di vincoli**.
Dato un vincolo $c \in \mathcal{C}$ e una nozione di *consistenza locale* (cfr. sezione 2.2), un **algoritmo di riduzione dei domini** rimuove valori 'incompatibili' con c dai domini delle variabili su cui è definito il vincolo.
Ad esempio, se $c \equiv x_i \neq 0$, il valore 0 dovrà essere rimosso da D_i.

Se un valore viene rimosso dal dominio di una variabile, l'effetto dev'essere *propagato* a tutti gli altri vincoli che 'condividono' quella variabile.
Ad esempio, se ho $c_1 \equiv x_i \neq 0$ e $c_2 \equiv x_i = x_j$ allora l'effetto della rimozione di 0 da D_i dev'essere propagato anche a D_j, in quanto nemmeno D_j potrà contenere il valore 0.
Questo processo iterativo continua per ogni vincolo, fino a che tutti i valori inconsistenti del CSP vengono rimossi dai corrispondenti domini: a questo punto, diremo che il CSP è **localmente consistente**.

Sfortunatamente, la consistenza locale non è sempre sufficiente per garantire la consistenza (*globale*) del CSP. Ad esempio, il processo di propagazione dei vincoli spesso non riesce ad individuare l'inconsistenza incontrata nell'osservazione 1.
In questi casi, per forzare la completezza è necessario l'utilizzo di strategie di ricerca non-deterministiche, quali ad esempio il **labeling**.
Come si vedrà più approfonditamente in seguito, questa tecnica permette di assegnare ad ogni variabile un valore del proprio dominio.
Tuttavia, a differenza del *Generate & Test* 'puro', il labeling utilizza *euristiche* che permettono ad esempio di scegliere in modo mirato quale valore assegnare ad una variabile (*value choice heuristic*) oppure l'ordine delle variabili sulle quali fare labeling (*variable choice heuristic*).
In questo modo si passa ad una strategia di ricerca 'informata' in uno spazio di ricerca ridotto dal processo di riduzione e propagazione.

Il costo totale della risoluzione è quindi determinato dalla somma tra il costo di ricerca della consistenza locale (tipicamente polinomiale) e quello di ricerca della consistenza globale (tipicamente esponenziale).

2.2 Nozioni di consistenza locale

Come detto, l'efficienza della risoluzione è in parte determinata dalla nozione di consistenza locale che viene applicata ai vincoli: rimane il problema di

decidere in quali casi applicare una nozione anziché un'altra.
Di seguito richiamiamo quattro definizioni di consistenza locale.

Definizione 2.6 (*Arc Consistency*)**.** Un vincolo binario $c(x_i, x_j)$ si dice **arc consistent** sse:

(i) per ogni $d_i \in D_i$ esiste un $d_j \in D_j$ tale che $(d_i, d_j) \in c$

(ii) per ogni $d_j \in D_j$ esiste un $d_i \in D_i$ tale che $(d_i, d_j) \in c$.

Definizione 2.7 (*Hyper-Arc Consistency*)**.** Un vincolo k-ario $c(x_{i_1}, \ldots, x_{i_k})$ si dice **hyper-arc consistent** sse per ogni $j \in \{1, \ldots, k\}$ e per ogni valore $d_{i_j} \in D_{i_j}$ esistono valori $d_{i_h} \in D_{i_h}$ con $h \in \{1, \ldots, k\} \setminus \{j\}$ tali che $(d_{i_1}, \ldots, d_{i_k}) \in c$.

Come si può notare dalla definizione, se $k = 2$ allora l'hyper-arc consistency (detta anche *k-consistency*) si riduce all'arc consistency.
Se $k = 1$ (cioè, il vincolo è unario) si parla invece di *node consistency*.

Si può inoltre osservare come tali nozioni di consistenza considerino *tutti* i possibili valori dei domini coinvolti: in particolari situazioni è possibile rilassare questa condizione.
Consideriamo infatti CSP in cui tutti i domini siano dotati di minimo e massimo (indicati con $min(D)$ e $max(D)$ rispettivamente) secondo una certa relazione d'ordine $\sqsubseteq$ su di un fissato universo $\mathcal{U}$.
Sotto queste condizioni, per ogni dominio $D \subseteq \mathcal{U}$ vale che:

$$D \subseteq [min(D)..max(D)] \stackrel{def}{=} \{x \in \mathcal{U} : min(D) \sqsubseteq x \sqsubseteq max(D)\}$$

Partendo da queste ipotesi sui domini, possiamo allora formulare le seguenti definizioni:

Definizione 2.8 (*Range Consistency*)**.** Un vincolo k-ario $c(x_{i_1}, \ldots, x_{i_k})$ si dice **range consistent** sse per ogni $j \in \{1, \ldots, k\}$ e per ogni valore $d_{i_j} \in D_{i_j}$ esistono valori $d_{i_h} \in [min(D_{i_h})..max(D_{i_h})]$ con $h \in \{1, \ldots, k\} \setminus \{j\}$ tali che $(d_{i_1}, \ldots, d_{i_k}) \in c$.

Definizione 2.9 (*Bound Consistency*)**.** Un vincolo k-ario $c(x_{i_1}, \ldots, x_{i_k})$ si dice **bound consistent** sse per ogni $j \in \{1, \ldots, k\}$ e per ogni valore $d_{i_j} \in \{min(D_{i_j}), max(D_{i_j})\}$ esistono valori $d_{i_h} \in [min(D_{i_h})..max(D_{i_h})]$ con $h \in \{1, \ldots, k\} \setminus \{j\}$ tali che $(d_{i_1}, \ldots, d_{i_k}) \in c$.

Si noti che queste due definizioni non considerano esattamente i valori dei vari domini, ma il più piccolo intervallo che li contiene, ossia la loro *chiusura convessa* $[min(D_{i_h})..max(D_{i_h})]$.

Si tratta quindi di una *sovra-approssimazione*: ciò ovviamente implica una perdita di precisione nel processo di riduzione, tipicamente contrastata da una maggiore efficienza. Questo è formalizzato qui di seguito.

Definizione 2.10 (*Consistenza Locale CSP*)**.** Un CSP $\mathcal{P}$ è rispettivamente range consistent, bound consistent o hyper-arc consistent se tutti i suoi vincoli lo sono.

Teorema 2.2.1 (*Relazioni tra consistency*)**.** *Sia $\mathcal{P}$ un CSP per cui sia possibile definire le nozioni di bound e range consistency. Allora vale che:*

$$\mathcal{P} \textit{ hyper-arc consistent} \implies \mathcal{P} \textit{ range consistent} \implies \mathcal{P} \textit{ bound consistent.}$$

Dimostrazione. Sia $\mathcal{P}$ = un CSP per cui valgano le condizioni della proposizione. Per ogni $i = 1, \ldots, n$ si ha che:

- $D_i \subseteq [min(D_i)..max(D_i)]$, perciò se $\mathcal{P}$ è hyper-arc consistent allora è anche range consistent
- $\{min(D_i), max(D_i)\} \subseteq D_i$, perciò se $\mathcal{P}$ è range consistent allora è anche bound consistent.

■

Capitolo 3

Intervalli e Multi-intervalli di interi

In questo capitolo verranno discussi in modo formale due *domini finiti* che verranno poi utilizzati per modellare il dominio delle variabili in CSP contenenti vincoli su interi.

3.1 Intervalli di interi

Definizione 3.1 (Intervallo di interi)**.** Sia $\mathbb{Z}$ l'insieme dei numeri interi e siano $a, b \in \mathbb{Z}$. Definiamo **intervallo (di interi)** di estremi a e b il *sottoinsieme finito* $[a..b] \subset \mathbb{Z}$ tale che:

$$[a..b] \stackrel{def}{=} \{x \in \mathbb{Z} : a \leq x \leq b\}$$

a è l'**estremo inferiore** (GLB, *Greatest Lower Bound*) mentre b è l'**estremo superiore** (LUB, *Least Upper Bound*) dell'intervallo.

Osservazione 2. Siano $a, b \in \mathbb{Z}$. Allora,

- Se $a > b$, si ha che $[a..b] = \varnothing$
- Se $a = b$, si ha che $[a..b] = [a..a] = \{a\}$
- Se $a \leq b$, si ha che $[a..b]$ ha esattamente $b - a + 1$ elementi.

Definizione 3.2 (Insieme $\mathbb{I}_\alpha$)**.** Sia $\alpha \geq 0$ una *fissata costante intera;*[1] d'ora in avanti indicheremo con $\mathbb{I}_\alpha$ l'insieme di tutti gli intervalli contenuti

[1] Intuitivamente, α assume la semantica della costante ∞, per cui si considera il suo valore come '*grande a piacere*'. Un approccio alternativo sarebbe quello di estendere la definizione di intervallo intero a $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{+\infty, -\infty\}$, tenendo conto della possibilità di forme indeterminate come $+\infty - \infty, \infty/\infty$ e $\infty \cdot 0$ e di intervalli illimitati.

nell'*universo* $\mathbb{Z}_\alpha \stackrel{def}{=} [-\alpha..\alpha]$, cioè:

$$\mathbb{I}_\alpha \stackrel{def}{=} \{[a..b] : a, b \in \mathbb{Z}_\alpha\}$$

Inoltre, preso un generico $I \in \mathbb{I}_\alpha \setminus \varnothing$, verrà indicato con:

- I^- il greatest lower bound di I
- I^+ il least upper bound di I.

Lemma 3.1.1. $\mathbb{I}_\alpha \subset \mathcal{P}(\mathbb{Z}_\alpha)$

Dimostrazione. Per definizione, $\mathbb{I}_\alpha \subseteq \mathbb{Z}_\alpha$. Esistono però elementi di $\mathcal{P}(\mathbb{Z}_\alpha)$ che non appartengono a $\mathbb{I}_\alpha$, basti pensare ad esempio all'insieme $\{-1, 1\}$.

■

Dal lemma segue quindi che $\mathbb{I}_\alpha$ non è isomorfo a $\mathcal{P}(\mathbb{Z}_\alpha)$; le conseguenze di tale risultato saranno affrontate nella prossima sezione.

3.1.1 Aritmetica degli intervalli

In questa sezione verranno esaminate alcune possibili operazioni su $\mathbb{I}_\alpha$ e le proprietà di cui esse godono. Prima però diamo alcune definizioni.

Definizione 3.3 (Normalizzazione)**.** Sia $A \subseteq \mathbb{Z}$. Definiamo operatore di **normalizzazione** (*rispetto a* $\mathbb{Z}_\alpha$) l'applicazione $\|\cdot\|_\alpha : \mathcal{P}(\mathbb{Z}) \longrightarrow \mathcal{P}(\mathbb{Z}_\alpha)$ tale che:

$$\|A\|_\alpha = A \cap \mathbb{Z}_\alpha$$

D'ora in avanti, la normalizzazione di un intervallo $[a..b]$ verrà indicata con $[\![a..b]\!]_\alpha$ anziché $\|[a..b]\|_\alpha$.

Definizione 3.4 (Chiusura convessa)**.** Sia $A \subseteq \mathbb{Z}$. Definiamo operatore di **chiusura convessa** (*rispetto a* $\mathbb{Z}_\alpha$) l'applicazione $\mathcal{CH}_\alpha : \mathcal{P}(\mathbb{Z}) \longrightarrow \mathbb{I}_\alpha$ tale che:

$$\mathcal{CH}_\alpha(A) \stackrel{def}{=} min_\subseteq\{I \in \mathbb{I}_\alpha : \|A\|_\alpha \subseteq I\}$$

Definizione 3.5 (Restrizione di un operatore a $\mathbb{I}_\alpha$)**.** Sia $\Phi : \mathcal{P}(\mathbb{Z})^k \longrightarrow \mathcal{P}(\mathbb{Z})$ un'operazione k-aria su $\mathcal{P}(\mathbb{Z})$. Definiamo **restrizione di** Φ **a** $\mathbb{I}_\alpha$ l'operazione $\Phi_\mathbb{I} : \mathbb{I}_\alpha^k \longrightarrow \mathbb{I}_\alpha$ tale che per ogni $I_1, \ldots, I_k \in \mathbb{I}_\alpha$

$$\Phi_\mathbb{I}(I_1, \ldots, I_k) \stackrel{def}{=} \mathcal{CH}_\alpha(\Phi(I_1, \ldots, I_k))$$

Definizione 3.6 (Operatori aritmetici su insiemi)**.** Siano $X, Y \subseteq \mathbb{Z}$.
Definiamo le operazioni di $+, -, \cdot, / : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \longrightarrow \mathcal{P}(\mathbb{Z})$ nel seguente modo:

$$X + Y \stackrel{def}{=} \{z \in \mathbb{Z} : (\exists x \in X)(\exists y \in Y)\ z = x + y\}$$
$$X - Y \stackrel{def}{=} \{z \in \mathbb{Z} : (\exists x \in X)(\exists y \in Y)\ z = x - y\}$$
$$X \cdot Y \stackrel{def}{=} \{z \in \mathbb{Z} : (\exists x \in X)(\exists y \in Y)\ z = x \cdot y\}$$
$$X/Y \stackrel{def}{=} \{z \in \mathbb{Z} : (\exists x \in X)(\exists y \in Y)\ x = z \cdot y\}$$

Come si può notare, le operazioni di $+$, $-$ e $\cdot$ sono definite in modo abbastanza naturale, estendendo le operazioni aritmetiche 'classiche' in $\mathbb{Z}$ all'insieme $\mathcal{P}(\mathbb{Z})$.

Per quel che riguarda $/$ invece il problema diventa più sottile, in quanto la divisione *tra interi* è un'operazione $/ : \mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) \longrightarrow \mathbb{Q}$.

Oltre al problema generale della divisione per zero, va perciò gestito il fatto che l'operazione $/$ *non è chiusa in* $\mathbb{Z}$: la divisione tra due interi non è necessariamente un intero.

Per ovviare a ciò, si potrebbe pensare di considerare $/$ come *divisione intera*, cioè tale che $x/y = \left\lfloor \frac{x}{y} \right\rfloor$, e definire quindi:

$$X/Y \stackrel{def}{=} \left\{ z \in \mathbb{Z} : (\exists x \in X)(\exists y \in Y)\ z = \left\lfloor \frac{x}{y} \right\rfloor \right\}$$

In questo modo tuttavia perderei importanti proprietà di equivalenza.
Ad esempio, in generale non varrebbe che $x/y = z/y \Rightarrow x = z$ con $y \neq 0$ (basti pensare che sarebbe $0/2 = 1/2 = 0$ ma ovviamente $0 \neq 1$).

Per questa ragione (e per altre che vedremo in seguito) si è deciso di mantenere la definizione 3.6, dalla quale segue la seguente osservazione:

Osservazione 3. Siano $X, Y, Z \subseteq \mathbb{Z}$. Si ha che:

$$Z = X + Y \iff X = Z - Y \iff Y = Z - X$$
$$Z = X \cdot Y \iff X = Z/Y \iff Y = Z/X$$

A questo punto, possiamo restringere le operazioni $+, -, \cdot, /$ a $\mathbb{I}_\alpha$. Si noti che anziché $+_{\mathbb{I}}, -_{\mathbb{I}}, \cdot_{\mathbb{I}}, /_{\mathbb{I}}$ tali operazioni verranno denotate con $\oplus, \ominus, \odot, \oslash$.

Osservazione 4 (Operazioni aritmetiche su intervalli). Siano $I, J \in \mathbb{I}_\alpha \setminus \varnothing$. Allora, è possibile restringere le operazioni $+, -, \cdot, /$ su $\mathcal{P}(\mathbb{Z})$ alle corrispondenti operazioni $\oplus, \ominus, \odot, \oslash : \mathbb{I}_\alpha \times \mathbb{I}_\alpha \longrightarrow \mathbb{I}_\alpha$ nel seguente modo:

- $I \oplus J = [\![I^- + J^- .. I^+ + J^+]\!]_\alpha$
- $I \ominus J = [\![I^- - J^+ .. I^+ - J^-]\!]_\alpha$
- $I \odot J = [\![min(A)..max(A)]\!]_\alpha$,
  con $A = \{I^- \cdot J^-, I^- \cdot J^+, I^+ \cdot J^-, I^+ \cdot J^+\} \subset \mathbb{Z}$
- $$I \oslash J = \begin{cases} \mathbb{Z}_\alpha & \text{se } 0 \in I \text{ e } 0 \in J \\ \varnothing & \text{se } 0 \notin I \text{ e } J = \{0\} \\ [-a..a], \text{ con a} = max(|I^-|, |I^+|) & \text{se } 0 \notin I \text{ e } \{-1, 1\} \subseteq J \\ I \oslash [1..J^+] & \text{se } 0 \notin I, \\ & J^- = 0 \text{ e } J^+ \neq 0 \\ I \oslash [J^- .. -1] & \text{se } 0 \notin I, \\ & J^+ = 0 \text{ e } J^- \neq 0 \\ [\![\lceil min(A) \rceil .. \lfloor max(A) \rfloor]\!]_\alpha, & \\ \text{con } A = \left\{ \frac{I^-}{J^-}, \frac{I^-}{J^+}, \frac{I^+}{J^-}, \frac{I^+}{J^+} \right\} \subset \mathbb{Q} & \text{se } 0 \notin J \end{cases}$$

Come si può notare, le operazioni aritmetiche sugli intervalli possono causare una notevole *perdita di informazione* causata dalla chiusura convessa. Si prenda ad esempio la moltiplicazione $[4..5] \odot [100..101] = [400..505]$; il risultato è un insieme di $505 - 400 + 1 = 106$ elementi, tuttavia i valori ammissibili di tale insieme sono solamente quattro: $400, 404, 500$ e 505.
Stesso discorso vale per la divisione tra intervalli.
Ad esempio, $\{1000\} \oslash \{2\} = [\lceil 2/1000 \rceil .. \lfloor 1000/2 \rfloor] = [1..500]$: risulta un insieme di 500 elementi anziché il singoletto $\{500\}$.

Inoltre, pur essendo gli intervalli insiemi di interi, anche le operazioni insiemistiche elementari ristrette agli intervalli non risultano particolarmente significative.
Intersezione a parte, per la quale per ogni $I, J \in \mathbb{I}_\alpha$ risulta che:

$$I \cap_{\mathbb{I}} J = I \cap J = [max(I^-, J^-)..min(I^+, J^+)]$$

per le altre operazioni insiemistiche 'classiche' si verificano perdite di informazione e proprietà elementari.

Infatti, per l'unione si ha che:

$$I \cup_{\mathbb{I}} J = [min(I^-, J^-)..max(I^+, J^+)]$$

Per cui, se per esempio $I = \{-1000\}$ e $J = \{1000\}$, si ha che $I \cup_{\mathbb{I}} J = [-1000..1000]$: anziché due soli valori l'insieme risultante ne contiene 2001.

Riguardo la differenza insiemistica, siano $I = [1..100]$ e $J = [20..30]$: applicando la definizione si ha che $I \setminus_{\mathbb{I}} J = \mathcal{CH}_\alpha(\{1, \ldots, 19, 31, \ldots, 100\}) = [1..100] = I$ pur non essendo $I \cap J = \varnothing$.

Inoltre, se $I = [-1..1]$, allora il suo complementare rispetto a $\mathbb{Z}_\alpha$ sarebbe $\sim_{\mathbb{I}} I = \mathcal{CH}_\alpha([-\alpha.. - 2] \cup [2..\alpha]) = [-\alpha..\alpha] = \mathbb{Z}_\alpha$ pur non essendo $I = \varnothing$.

In conclusione, al contrario di $\mathcal{P}(\mathbb{Z})$, $\mathbb{I}_\alpha$ non gode di proprietà algebriche particolarmente interessanti.
Tuttavia, il concetto di intervallo può tornare utile in vari casi per *approssimare* il valore di una certa variabile attraverso un *range* di valori che essa può assumere: è il caso ad esempio dei domini delle variabili nel CLP($\mathcal{FD}$).

3.2 Multi-intervalli

In questa sezione introdurremo una generalizzazione del precedente concetto di intervallo, che avrà lo scopo di superare le limitazioni viste fino ad ora.

Per cominciare, definiamo una relazione $\prec \subseteq (\mathbb{I}_\alpha \setminus \varnothing) \times (\mathbb{I}_\alpha \setminus \varnothing)$ tale che per ogni $I, J \in \mathbb{I}_\alpha$

$$I \prec J \overset{def}{\iff} I^+ < J^- - 1$$

Proposizione 3.2.1. *$\prec$ è una relazione di **ordine stretto** ma **non totale** sull'insieme $\mathbb{I}_\alpha \setminus \varnothing$.*

Dimostrazione. $\prec$ è relazione di ordine stretto in quanto:

- *Antiriflessiva:* per ogni $I \in \mathbb{I}_\alpha \setminus \varnothing$, ho che $I^+ \geq I^- > I^- - 1$ da cui per transitività $I^+ > I^- - 1$ quindi $I \not\prec I$.
- *Transitiva:* siano $I \prec J$ e $J \prec K$. Allora per definizione $I^+ < J^- - 1$ e $J^+ < K^- - 1$; inoltre, essendo J non vuoto si ha che $J^- \leq J^+$ perciò $I^+ < J^- - 1 < J^- \leq J^+ < K^- - 1$. Per transitività, $I^+ < K^- - 1$ quindi $I \prec K$.

Infine, $\prec$ non è totale: presi ad esempio $I = \{0\}$ e $J = \{1\}$ è banale osservare che $I \not\prec J$, $J \not\prec I$ e $I \neq J$.

■

Definizione 3.7 (Multi-intervallo)**.** Un **multi-intervallo** (di interi) di *ordine* $n \geq 0$ è un sottoinsieme $M \subset \mathbb{Z}$ definito da una n-upla di intervalli non vuoti $I_1, I_2, \ldots, I_n \in \mathbb{I}_\alpha \setminus \varnothing$ tali che:

- $M = I_1 \cup I_2 \cup \ldots \cup I_n$
- $I_1 \prec I_2 \prec \ldots \prec I_n$

$M^- \stackrel{def}{=} I_1^-$ è il GLB del multi-intervallo mentre $M^+ \stackrel{def}{=} I_n^+$ è il suo LUB.

D'ora in avanti, indicheremo con $\mathbb{M}_\alpha$ l'insieme di tutti i multi-intervalli contenuti in $\mathbb{Z}_\alpha$.

Inoltre, un multi-intervallo definito dagli intervalli $I_1, \ldots, I_n$ con $n > 1$ verrà denotato con la notazione $\{I_1^-..I_1^+, \ldots, I_n^-..I_n^+\}$

Osservazione 5. Sia M un multi-intervallo di ordine n. Allora,

- $n = 0 \iff M = \varnothing$
- $n = 1 \iff M \in \mathbb{I}_\alpha$
- $|M| = \sum_{k=1}^{n} |I_k|$

Teorema 3.2.2.

$$\mathbb{M}_\alpha = \mathcal{P}(\mathbb{Z}_\alpha)$$

Dimostrazione. Per definizione, preso un generico $M \in \mathbb{M}_\alpha$ si ha che $M \subseteq \mathbb{Z}_\alpha$ perciò $M \in \mathcal{P}(\mathbb{Z}_\alpha)$ da cui segue che $\mathbb{M}_\alpha \subseteq \mathcal{P}(\mathbb{Z}_\alpha)$ per genericità di M. Per completare la dimostrazione, rimane da verificare che $\mathcal{P}(\mathbb{Z}_\alpha) \subseteq \mathbb{M}_\alpha$.

Sia allora $A \in \mathcal{P}(\mathbb{Z}_\alpha)$ generico; tale A sarà una *catena finita* di n elementi di $\mathbb{Z}$, con $0 \leq n \leq 2\alpha + 1$. Si consideri quindi l'ordinamento crescente dei suoi elementi $\langle a_1, \ldots, a_n \rangle$, tale che $a_1 < \ldots < a_n$.
Definiamo allora una relazione $\equiv \subseteq A \times A$ tale che, per ogni $x, y \in A$,

$$x \equiv y \iff^{def} [min(x, y)..max(x, y)] \subseteq A$$

Si dimostra facilmente che tale relazione è di equivalenza. Inoltre, $\equiv$ è a *classi (fortemente) separate:*

$$[x] \neq [y] \wedge x < y \wedge x' \in [x] \wedge y' \in [y] \Longrightarrow x' < y' - 1$$

Supponiamo infatti per assurdo che $x' \not< y$, allora sarebbe $x < y \leq x'$ quindi $x \equiv y$ contro l'ipotesi che $[x] \neq [y]$. Perciò, $x' < y$. Ragionando in modo analogo, se per assurdo $x' \not< y' - 1$ varrebbe che $y' - 1 \leq x' < y$ da cui $x' \equiv y$: assurdo. Quindi, $x' < y' - 1$.

Grazie a questo risultato, possiamo notare come l'insieme quoziente $A/_{\equiv}$ sia composto da una catena (secondo $\prec$) di $m \leq n$ intervalli non vuoti $I_1, \ldots, I_m$. Allora, $M = I_1 \cup \ldots \cup I_m$ è un multi-intervallo ed essendo tali intervalli una partizione di A si ha che $I_1 \cup \ldots \cup I_m = A$.
Perciò, $A \in \mathbb{M}_\alpha$: dalla genericità di A segue l'inclusione $\mathcal{P}(\mathbb{Z}_\alpha) \subseteq \mathbb{M}_\alpha$ e quindi la tesi.

■

Questo importante risultato stabilisce quindi un *isomorfismo* tra $\mathcal{P}(\mathbb{Z}_\alpha)$ e $\mathbb{M}_\alpha$, che permetterà d'ora in avanti di considerare algebricamente equivalenti i multi-intervalli di $\mathbb{M}_\alpha$ e gli insiemi di interi di $\mathcal{P}(\mathbb{Z}_\alpha)$.

Corollario 3.2.3. $\mathbb{I}_\alpha \subset \mathbb{M}_\alpha$

Dimostrazione. Segue banalmente dal teorema e dal lemma 3.1.1.

■

Il corollario afferma che un intervallo è di fatto un particolare multi-intervallo *degenere* (di ordine minore o uguale a uno, come già osservato in 5).

3.2.1 Aritmetica dei multi-intervalli

Come dimostrato, $\mathbb{M}_\alpha$ e $\mathcal{P}(\mathbb{Z}_\alpha)$ sono di fatto lo stesso insieme. Quindi, non c'è alcun bisogno di utilizzare operatori di chiusura come in 3.4 per restringere gli operatori su $\mathcal{P}(\mathbb{Z})$ a $\mathbb{M}_\alpha$.
Tuttavia, rimane il problema della *normalizzazione* rispetto a $\mathbb{Z}_\alpha$.

Definizione 3.8 (Restrizione di un operatore a $\mathbb{M}_\alpha$)**.** Si consideri $\Phi : \mathcal{P}(\mathbb{Z})^k \longrightarrow \mathcal{P}(\mathbb{Z})$ un operazione k-aria su $\mathcal{P}(\mathbb{Z})$. Definisco **restrizione di Φ a $\mathbb{M}_\alpha$** l'operazione $\Phi_\mathbb{M} : \mathbb{M}_\alpha^k \longrightarrow \mathbb{M}_\alpha$ tale che per ogni $I_1, \ldots, I_k \in \mathbb{M}_\alpha$

$$\Phi_\mathbb{M}(I_1, \ldots, I_k) \stackrel{def}{=} \|\Phi(I_1, \ldots, I_k)\|_\alpha$$

Con questa definizione, le operazioni fra multi-intervalli superano molti problemi di potere espressivo e perdita di informazione riscontrati nelle operazioni fra intervalli.

Consideriamo ad esempio le operazioni aritmetiche sui multi-intervalli. Come per gli intervalli, definiamo prima di tutto una nuova notazione che permetterà di alleggerire la lettura: d'ora in avanti, per indicare le restrizioni di $+, -, \cdot, /$ a $\mathbb{M}_\alpha$, utilizzeremo i simboli $\boxplus, \boxminus, \boxdot, \boxslash$ anziché $+_\mathbb{M}, -_\mathbb{M}, \cdot_\mathbb{M}, /_\mathbb{M}$.
Vediamo quindi come si comportano tali operatori.

Si prenda come esempio l'addizione $[-1..1] \boxplus \{5, 10..15\} = \{4..6, 9..16\}$: questa operazione non è consentita in $\mathbb{I}_\alpha$, non essendo $\{5, 10..15\}$ un intervallo. Discorso analogo vale per la sottrazione $\boxminus$.

La moltiplicazione $[4..5] \boxdot [100..101]$ vista in precedenza con gli intervalli risulta invece $\{400, 404, 500, 505\}$: ottengo un insieme vistosamente ridotto rispetto a $[400..505] = [4..5] \odot [100..101]$.

Per quel che riguarda la divisione rimane il problema della non-chiusura di / in $\mathbb{Z}$. Tuttavia, con la definizione di multi-intervallo è possibile restringere l'insieme divisore eventualmente escludendo lo zero.
Ad esempio, al contrario di quanto possa avvenire in $\mathbb{I}_\alpha$, nel dominio $\mathbb{M}_\alpha$ posso eseguire la divisione $[0..1] \boxslash \{-1, 1\} = [-1..1]$. In $\mathbb{I}_\alpha$ potrei al limite effettuare la divisione $[0..1] \oslash [-1..1] = \mathbb{Z}_\alpha$: la differenza fra i due risultati è evidente.

Inoltre, come ulteriore conseguenza del teorema 3.2, le operazioni insiemistiche 'classiche' ristrette a $\mathbb{M}_\alpha$ acquisiscono consistenza.
Infatti, la restrizione di un operatore $* \in \{\sim, \cap, \cup, \setminus\}$ non necessiterà di alcuna normalizzazione, essendo tali operazioni *chiuse* nell'universo $\mathbb{Z}_\alpha$.
Per questo motivo, anziché utilizzare i simboli $\sim_{\mathbb{M}}, \cap_{\mathbb{M}}, \cup_{\mathbb{M}}, \setminus_{\mathbb{M}}$ manterremo la corrispondente notazione $\sim, \cap, \cup, \setminus$.

Soffermiamoci infine sul principale problema della normalizzazione rispetto a $\mathbb{Z}_\alpha$, ossia la *finitezza* di tale universo.

Osservazione 6 (Finitezza di $\mathbb{Z}_\alpha$). L'utilizzo di un **universo finito** $\mathbb{Z}_\alpha$ (pur grande a piacere) pone delle limitazioni nelle operazioni fra multi-intervalli. Ad esempio:

- $\mathbb{Z}_\alpha \boxplus \mathbb{Z}_\alpha = \mathbb{Z}_\alpha \boxminus \mathbb{Z}_\alpha = [\![-2\alpha..2\alpha]\!]_\alpha = \mathbb{Z}_\alpha$
- $\mathbb{Z}_\alpha \boxdot \mathbb{Z}_\alpha = [\![-\alpha^2..\alpha^2]\!]_\alpha = \mathbb{Z}_\alpha$
- $\{0\} \boxslash \{0\} = \mathbb{Z}_\alpha$

A corollario di ciò, si può notare come anche per i multi-intervalli la normalizzazione possa comportare una perdita di informazione.

Considerando gli esempi precedenti infatti si può osservare come nella somma e nella sottrazione vengano 'scartati' 2α valori corretti appartenenti all'insieme

$$(\mathbb{Z}_\alpha + \mathbb{Z}_\alpha) \setminus (\mathbb{Z}_\alpha \boxplus \mathbb{Z}_\alpha) = \{-2\alpha, \ldots, -\alpha - 1, \alpha + 1, \ldots, 2\alpha\}$$

Analogamente, per il prodotto perdo addirittura $2\alpha^2 - 2\alpha$ valori appartenenti all'insieme

$$(\mathbb{Z}_\alpha \cdot \mathbb{Z}_\alpha) \setminus (\mathbb{Z}_\alpha \boxdot \mathbb{Z}_\alpha) = \{-\alpha^2, \ldots, -\alpha - 1, \alpha + 1, \ldots, \alpha^2\}$$

La divisione consentirebbe invece una 'strana' operazione del tipo $\{0\}/\{0\}$. Ad ogni modo, seguendo la definizione data di divisione insiemistica, non vengono considerati un *infinità numerabile* di valori appartenenti all'insieme

$$(\{0\}/\{0\}) \setminus (\{0\} \boxslash \{0\}) = (-\infty, -\alpha - 1] \cup [\alpha + 1, +\infty)$$

In conclusione, è bene scegliere in modo opportuno il valore della costante α: un valore troppo piccolo comporterebbe un universo ristretto e poco significativo; viceversa, un valore troppo grande potrebbe causare (oltre alla suddetta perdita di informazione) comportamenti indesiderati nel calcolo computazionale, come ad esempio l'*overflow.*

Capitolo 4

Risoluzione di vincoli su interi

In questo capitolo, dopo aver definito formalmente il linguaggio $\mathcal{L}_{\mathcal{FD}}$ e una sua interpretazione, verranno descritti in termini astratti regole e algoritmi per la risoluzione di CSP basati su tale linguaggio.

4.1 Il linguaggio $\mathcal{L}_{\mathcal{FD}}$

In questa sezione verrà definito formalmente il linguaggio $\mathcal{L}_{\mathcal{FD}}$, che servirà in seguito per trattare vincoli su variabili di tipo intero, aventi come dominio l'insieme $\mathbb{Z}_\alpha$.

Definizione 4.1 (Alfabeto di $\mathcal{L}_{\mathcal{FD}}$)**.** L'**alfabeto** del linguaggio $\mathcal{L}_{\mathcal{FD}}$ è costituito da:

- un insieme *infinito* di **variabili** $\mathcal{V}_{\mathcal{FD}} \stackrel{def}{=} \{x, y, z, \ldots\}$
- il simbolo di **congiunzione logica** $\wedge$
- le **parentesi** tonde (e)
- il predicato binario di **dominio** ::
- un insieme di **predicati binari** $\mathcal{P}_{\mathcal{FD}} \stackrel{def}{=} \{=, \neq, <, \leq, >, \geq\}$
- un insieme di **simboli funzionali binari** $\mathcal{F}_{\mathcal{FD}} \stackrel{def}{=} \{+, -, \cdot, /\}$
- un insieme *numerabile* di **costanti** $\mathcal{Z}_{\mathcal{FD}} \stackrel{def}{=} \{0, (-1), 1, (-2), 2, \ldots\}$.

Definizione 4.2 (Termini di $\mathcal{L}_{\mathcal{FD}}$)**.** L'insieme $\mathcal{T}_{\mathcal{FD}}$ dei **termini** del linguaggio $\mathcal{L}_{\mathcal{FD}}$ è così definito:

(i) ogni variabile è un termine: $\mathcal{V}_{\mathcal{FD}} \subseteq \mathcal{T}_{\mathcal{FD}}$

(ii) ogni costante è un termine: $\mathcal{Z}_{\mathcal{FD}} \subseteq \mathcal{T}_{\mathcal{FD}}$

(iii) se $t_1, t_2 \in \mathcal{T}_{\mathcal{FD}}$ e $* \in \mathcal{F}_{\mathcal{FD}}$, allora $(t_1 * t_2) \in \mathcal{T}_{\mathcal{FD}}$

(iv) nient'altro appartiene a $\mathcal{T}_{\mathcal{FD}}$.

Definizione 4.3 (Vincoli atomici di $\mathcal{L}_{\mathcal{FD}}$)**.** L'insieme $\mathcal{AC}_{\mathcal{FD}}$ dei **vincoli atomici** del linguaggio $\mathcal{L}_{\mathcal{FD}}$ è così definito:

(i) se $t \in \mathcal{T}_{\mathcal{FD}}$ e $M \in \mathbb{M}_\alpha$ allora $t :: M \in \mathcal{AC}_{\mathcal{FD}}$

(ii) se $t_1, t_2 \in \mathcal{T}_{\mathcal{FD}}$ e $* \in \mathcal{P}_{\mathcal{FD}}$ allora $t_1 * t_2 \in \mathcal{AC}_{\mathcal{FD}}$

(iii) nient'altro appartiene a $\mathcal{AC}_{\mathcal{FD}}$.

Definizione 4.4 (Vincoli di $\mathcal{L}_{\mathcal{FD}}$)**.** L'insieme $\mathcal{C}_{\mathcal{FD}}$ dei **vincoli** del linguaggio $\mathcal{L}_{\mathcal{FD}}$ è così definito:

(i) ogni vincolo atomico è un vincolo: $\mathcal{AC}_{\mathcal{FD}} \subseteq \mathcal{C}_{\mathcal{FD}}$

(ii) se $c_1, c_2 \in \mathcal{C}_{\mathcal{FD}}$ allora $c_1 \wedge c_2 \in \mathcal{C}_{\mathcal{FD}}$

(iii) nient'altro appartiene a $\mathcal{C}_{\mathcal{FD}}$.

Esempio 1 (Termini e vincoli $\mathcal{L}_{\mathcal{FD}}$). Riportiamo a titolo esplicativo una piccola casistica di elementi che (non) appartengono agli insiemi sopracitati. Sono ad esempio termini le seguenti espressioni:[1]

- $x + 3$
- $(1 - y) \cdot (-2)$
- $z/(-4) + 1$

mentre non lo sono:

- $x \leq y + 1$
- $x - x = 0$

[1]D'ora in avanti, per alleggerire la lettura, verranno omesse le parentesi 'non significative': per evitare ambiguità si utilizzerà l'ordine di precedenza 'classico' degli operatori di $\mathcal{F}_{\mathcal{FD}}$, associando a sinistra.

Sono invece vincoli le seguenti espressioni:

- $x :: \{1..10, 20..100\}$
- $x \cdot x + y \cdot y = 1 \wedge x \neq -1$
- $z \geq 0 \wedge z < 0 \wedge z = 0$

4.1.1 Semantica del linguaggio $\mathcal{L}_{\mathcal{FD}}$

Per conferire significato ai simboli di $\mathcal{L}_{\mathcal{FD}}$, utilizzeremo un'*interpretazione* 'naturale' Δ su $\mathbb{Z}$ che associa:

- ai predicati $P \in \mathcal{P}_{\mathcal{FD}}$, la relazione $P^\Delta \subseteq \mathbb{Z} \times \mathbb{Z}$ tale che $P^\Delta \stackrel{def}{=} P \restriction_{\mathbb{Z}\times\mathbb{Z}}$
- ai simboli funzionali $f \in \mathcal{F}_{\mathcal{FD}} \setminus \{/\}$, l'operazione $f^\Delta : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ tale che $f^\Delta \stackrel{def}{=} f \restriction_{\mathbb{Z}\times\mathbb{Z}}$
- al simbolo funzionale / l'operazione $/^\Delta : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ tale che per ogni $x, y \in \mathbb{Z}$

  $$x /^\Delta y \stackrel{def}{=} \begin{cases} \dfrac{x}{y}, & \text{se } \dfrac{x}{y} \in \mathbb{Z}_\alpha \\ \alpha + 1 & \text{altrimenti} \end{cases}$$

  In questo modo, $x /^\Delta y \in \mathbb{Z}_\alpha \iff \dfrac{x}{y} \in \mathbb{Z}_\alpha$.
- alle costanti $k \in \mathcal{Z}_{\mathcal{FD}}$, il numero intero $k^\Delta \stackrel{def}{=} k \in \mathbb{Z}$.

Fissata questa interpretazione, possiamo definire formalmente il concetto di *soddisfacimento di vincoli*.

Definizione 4.5 (Funzione di assegnazione)**.** Sia $\mathbb{Z}_\alpha$ l'insieme universo definito in 3.2. Si definisce (*funzione di*) **assegnazione** su $\mathcal{V}_{\mathcal{FD}}$ una qualsiasi funzione $\varphi : \mathcal{V}_{\mathcal{FD}} \longrightarrow \mathbb{Z}_\alpha$. Una volta definita tale φ, è possibile ricavarne la sua **estensione ai termini** $\varphi^\Delta : \mathcal{T}_{\mathcal{FD}} \longrightarrow \mathbb{Z}$ tale che:

$$\varphi^\Delta(x) \stackrel{def}{=} \varphi(x), \qquad \text{per ogni } x \in \mathcal{V}_{\mathcal{FD}}$$
$$\varphi^\Delta(k) \stackrel{def}{=} k^\Delta, \qquad \text{per ogni } k \in \mathcal{Z}_{\mathcal{FD}}$$
$$\varphi^\Delta(t_1 * t_2) = \varphi^\Delta(t_1) *^\Delta \varphi^\Delta(t_2), \qquad \text{per ogni } t_1, t_2 \in \mathcal{T}_{\mathcal{FD}} \text{ e per ogni } * \in \mathcal{F}_{\mathcal{FD}}$$

Definizione 4.6 (Soddisfacimento di vincoli)**.** Sia $c \in \mathcal{C}_{\mathcal{FD}}$ e φ una funzione di assegnazione su $\mathcal{V}_{\mathcal{FD}}$. Definiamo formalmente la locuzione 'c **è soddisfatto da** φ', che abbrevieremo con:

$$\models_\varphi c$$

nel seguente modo:

$$\begin{array}{lcll} \models_\varphi t :: M & \stackrel{def}{\Longleftrightarrow} & \varphi^\Delta(t) \in M, & \text{per ogni } t \in \mathcal{T}_{\mathcal{FD}}, M \in \mathbb{M}_\alpha \\ \models_\varphi t_1 * t_2 & \stackrel{def}{\Longleftrightarrow} & \varphi^\Delta(t_1) *^\Delta \varphi^\Delta(t_2), & \text{per ogni } t_1, t_2 \in \mathcal{T}_{\mathcal{FD}}, * \in \mathcal{P}_{\mathcal{FD}} \\ \models_\varphi c_1 \wedge \ldots \wedge c_k & \stackrel{def}{\Longleftrightarrow} & \underset{i=1,\ldots,k}{\forall} \models_\varphi c_i, & \text{per ogni } c_1, \ldots, c_k \in \mathcal{C}_{\mathcal{FD}} \end{array}$$

Diremo quindi che un vincolo $c \in \mathcal{C}_{\mathcal{FD}}$ (non) è **soddisfacibile** se (non) esiste un'assegnazione $\varphi : \mathcal{V}_{\mathcal{FD}} \longrightarrow \mathbb{Z}_\alpha$ che lo soddisfa, cioè tale che $\models_\varphi c$.

Esempio 2 (Soddisfacimento di vincoli)*.* Consideriamo il vincolo:

$$x \leq 10 \wedge x :: \{-5..15, 20\}$$

Tale vincolo è soddisfacibile, basta prendere un'assegnazione φ tale che $-5 \leq \varphi(x) \leq 10$. Ad esempio, presa φ tale che $\varphi(x) = 0$ si ha che:[2]

$$\begin{array}{rcl} \models_\varphi x \leq 10 \wedge x :: \{-5, 20\} & \Longleftrightarrow & \models_\varphi x \leq 10 \quad \text{e} \quad \models_\varphi x :: \{-5..15, 20\} \\ & \Longleftrightarrow & \varphi^\Delta(x) \leq 10 \quad \text{e} \quad \varphi^\Delta(x) \in \{-5..15, 20\} \\ & \Longleftrightarrow & \varphi(x) \leq 10 \quad \text{e} \quad \varphi(x) \in \{-5..15, 20\} \\ & \Longleftrightarrow & 0 \leq 10 \quad \text{e} \quad 0 \in \{-5..15, 20\} \end{array}$$

Viceversa, si prenda il vincolo $x = \alpha + 1$. Nell'interpretazione che abbiamo fornito, tale vincolo non è soddisfacibile. Infatti, non esiste alcuna $\varphi : \mathcal{V}_{\mathcal{FD}} \longrightarrow \mathbb{Z}_\alpha$ tale che $\varphi^\Delta(x) = \varphi^\Delta(\alpha + 1) = \varphi^\Delta(\alpha) +^\Delta \varphi^\Delta(1) = \alpha + 1$, in quanto per definizione $\varphi^\Delta(x) = \varphi(x) \in \mathbb{Z}_\alpha$ e $max_\leq(\mathbb{Z}_\alpha) = \alpha$.
Quindi, anche un vincolo del tipo $z = x/y$ con $\dfrac{\varphi(x)}{\varphi(y)} \notin \mathbb{Z}_\alpha$ non sarà soddisfacibile, essendo che:

$$\varphi^\Delta(x/y) = \varphi^\Delta(x) \ /^\Delta \ \varphi^\Delta(y) = \varphi(x) \ /^\Delta \ \varphi(y) = \alpha + 1$$

.

[2]Laddove non crei ambiguità, d'ora in avanti al posto della notazione $a *^\Delta b$ si userà la più naturale notazione $a * b$.

Definizione 4.7 (Conseguenza ed equivalenza logica)**.** Un *insieme* di vincoli $\Gamma \subseteq \mathcal{C}_{\mathcal{FD}}$ si dice soddisfacibile se esiste un'assegnazione φ che soddisfa Γ (cioè, soddisfa *tutti* i suoi vincoli).
Un vincolo $c \in \mathcal{C}_{\mathcal{FD}}$ si dice **conseguenza logica** di un insieme di vincoli $\Gamma \subseteq \mathcal{C}_{\mathcal{FD}}$, e si indica con

$$\Gamma \models c$$

se ogni assegnazione che soddisfa Γ soddisfa anche c.
Due vincoli c_1 e c_2 sono **logicamente equivalenti** sse $c_1 \models c_2$ e $c_2 \models c_1$.[3]
Generalizzando, due *insiemi* di vincoli Γ, Δ sono logicamente equivalenti sse per ogni $c \in \Gamma, d \in \Delta$ si ha che $\Gamma \models d$ e $\Delta \models c$.
Per indicare due (insiemi di) vincoli logicamente equivalenti C e D si userà la notazione

$$C \models\!\models D$$

4.2 Risoluzione dei vincoli di $\mathcal{L}_{\mathcal{FD}}$

Dopo aver definito il linguaggio $\mathcal{L}_{\mathcal{FD}}$ e la sua semantica, passiamo ora a descrivere problemi di soddisfacimento di vincoli basati su tale linguaggio.

Definizione 4.8 ($CSP(\mathcal{FD})$)**.** Un CSP basato su $\mathcal{L}_{\mathcal{FD}}$, in breve $CSP(\mathcal{FD})$, è un CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ nel quale:

- $\mathcal{V} = \{x_1, \ldots, x_n\} \subseteq \mathcal{V}_{\mathcal{FD}}$
- $\mathcal{D} = D_1 \times \ldots \times D_n$ con $D_i \in \mathbb{M}_\alpha$ e $x_i :: D_i$ per ogni $i = 1, \ldots, n$
- $\mathcal{C} = \{c_1, \ldots, c_m\} \subseteq \mathcal{C}_{\mathcal{FD}}$

L'insieme $\mathcal{C}$ è detto anche **constraint store** di $\mathcal{P}$

Per alleggerire la notazione, d'ora in avanti (a meno di ambiguità) le variabili di $\mathcal{V}$ verranno indicate con le meta-variabili $x, y, z, \ldots$ anziché $x_1, x_2, x_3, \ldots$; i corrispondenti domini saranno denotati con $D_x, D_y, D_z, \ldots$

Definizione 4.9 (Constraint store semplificato)**.** Sia $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ un $CSP(\mathcal{FD})$. Dal constraint store $\mathcal{C}$ è possibile ottenere un insieme di vincoli atomici *logicamente equivalente* $\mathcal{CS} \subseteq \mathcal{AC}_{\mathcal{FD}}$ tale che ogni vincolo $a \in \mathcal{CS}$ è della forma:

(i) $x :: M$

(ii) $x * y$

[3] Con un abuso di notazione, se $c, c' \in \mathcal{C}_{\mathcal{FD}}$ si scriverà $c \models c'$ anziché $\{c\} \models c'$

(iii) $x * k$

(iv) $z = x + y$

(v) $z = x + k$

(vi) $z = x \cdot y$

(vii) $z = x \cdot k$

dove $M \in \mathbb{M}_\alpha$, $x, y, z \in \mathcal{V}_{\mathcal{FD}}$, $k \in \mathcal{Z}_{\mathcal{FD}}$ e $* \in \{=, \neq, <, \leq\}$.
L'insieme $\mathcal{CS}$ verrà detto **constraint store semplificato** di $\mathcal{P}$.

Osservazione 7 (Semplificazione del constraint store). Senza inoltrarci in laboriose dimostrazioni, si può notare come sia sempre possibile semplificare $\mathcal{C}$ in un constraint store logicamente equivalente $\mathcal{CS}$, infatti:

- ogni vincolo non atomico della forma $c_1 \wedge \ldots \wedge c_k$ verrà 'espanso' considerando i singoli vincoli $c_1, \ldots, c_k$ ed eventualmente ripetendo ricorsivamente il procedimento su di essi, fino ad ottenere soli vincoli atomici logicamente equivalenti $a_1, \ldots, a_h$, con $h \geq k$
- i vincoli di $>$ e $\geq$ verranno riscritti in forma di $<$ e $\leq$ rispettivamente, infatti:

$$t_1 > t_2 \models\!\!\models t_2 < t_1$$
$$t_1 \geq t_2 \models\!\!\models t_2 \leq t_1$$

- i vincoli di $-$ e $/$ verranno riscritti in forma di $+$ e $\cdot$ rispettivamente, infatti:

$$t_1 = t_2 - t_3 \models\!\!\models t_2 = t_1 + t_3$$
$$t_1 = t_2/t_3 \models\!\!\models \{t_2 = t_1 \cdot t_3, t_3 \neq 0\}$$

- se un vincolo non è della forma **(iii)**, **(v)** o **(vii)**, allora ogni sua costante $k \in \mathcal{Z}_{\mathcal{FD}}$ sarà sostituita da una *nuova variabile ausiliaria* $x_{(k)}$ e verranno aggiunti i corrispondenti vincoli di uguaglianza $x_{(k)} = k$
- ogni vincolo di uguaglianza verrà riscritto in una forma logicamente equivalente tra $x = y$, **(iv)**, ..., **(vii)**, eventualmente aggiungendo nuovi vincoli e variabili ausiliarie
- ogni vincolo di disuguaglianza verrà riscritto in una forma logicamente equivalente tra $x \neq y$, $x < y$ e $x \leq y$, eventualmente aggiungendo nuovi vincoli e variabili ausiliarie.

Ad esempio, se

$$\mathcal{C} = \{x + y \geq 3 - z,\ y \neq x + 1,\ v = u/2 \wedge u :: \{1..10\}\}$$

allora il corrispondente constraint store semplificato sarà della forma:

$$\mathcal{CS} = \{x_1 = 3,\ x_2 = x + y,\ x_1 = x_3 + z,\ x_2 \leq x_3,\ x_4 = 1,\ x_5 = x + x_4,\ y \neq x_5,\ u = v \cdot 2,\ 2 \neq 0,\ u :: \{1..10\}\}$$

Teorema 4.2.1 (Consistenza e soddisfacibilità)**.** *Sia $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ un $CSP(\mathcal{FD})$ e $\mathcal{CS}$ il corrispondente constraint store semplificato. Allora,*

$$\mathcal{P} \textit{ consistente} \iff \mathcal{CS} \textit{ soddisfacibile}$$

Dimostrazione. Per l'osservazione precedente, è sufficiente mostrare che $\mathcal{P}$ consistente $\iff$ $\mathcal{C}$ soddisfacibile, essendo $\mathcal{C} \models\!\!\models \mathcal{CS}$.
Se $\mathcal{P}$ è consistente, per definizione esiste una n-upla $d = \langle d_1, \ldots, d_n \rangle \in \mathcal{D}$ che soddisfa ogni $c \in \mathcal{C}$.
Quindi, presa $\varphi : \mathcal{V}_{\mathcal{FD}} \longrightarrow \mathbb{Z}_\alpha$ tale che $\varphi(x_i) = d_i$ per ogni $i = 1, \ldots, n$ si ha che φ soddisfa ogni $c \in \mathcal{C}$ perciò $\mathcal{C}$ è soddisfacibile.

Viceversa, se $\mathcal{C}$ è soddisfacibile esiste un'assegnazione φ che lo soddisfa. Presa allora la n-upla $d = \langle \varphi(x_1), \ldots, \varphi(x_n) \rangle$ si ha che d soddisfa ogni $c \in \mathcal{C}$ quindi $\mathcal{P}$ è consistente.

■

Dalla proposizione segue che un $CSP(\mathcal{FD})$ ha (almeno) una soluzione se e soltanto se il corrispondente $\mathcal{CS}$ è soddisfacibile, cioè esiste un'assegnazione di valori alle variabili di $\mathcal{CS}$ che soddisfa ogni suo vincolo atomico.

A questo punto, dato un $CSP(\mathcal{FD})$, ci possiamo chiedere quali criteri applicare per trovare (se esiste) una sua soluzione (o tutte le sue soluzioni).

Come accennato nel capitolo 2 l'approccio *'Generate & Test'*, seppur completo, è altamente inefficiente. Infatti, le possibili funzioni di assegnazione sono esattamente $\prod_{i=1}^{n} |D_i|$.

Per questo motivo, verranno utilizzate apposite *regole di riduzione dei domini* da applicare ad ogni vincolo atomico $a \in \mathcal{CS}$.
Inoltre, a ciascuno di questi vincoli sarà associata una nozione di *consistenza locale* (cfr. sezione 2.2). La scelta della nozione da utilizzare sarà un compromesso tra potere riduttivo sui domini ed efficienza della sua implementazione.

In seguito verranno presentate ciascuna di queste regole, la cui forma generale è formalizzata nel seguente modo:

Definizione 4.10 (Regola di riduzione)**.** Sia $a(x_{i_1}, \ldots, x_{i_k}) \in \mathcal{CS}$ un vincolo atomico definito sulle variabili $x_{i_1}, \ldots, x_{i_k} \in \mathcal{V}_{\mathcal{FD}}$.
Una **regola di riduzione** dei domini per a è un'operazione R_a che permette, nel caso sia soddisfatta un'*eventuale pre-condizione* P, di 'restringere' ogni dominio D_{i_j} calcolando un *nuovo dominio* $D'_{i_j} \subseteq D_{i_j}$ ottenuto intersecando D_{i_j} con un opportuno insieme di valori $\Theta_{i_j} \subseteq \mathbb{R}$.
Per indicare una tale regola useremo la notazione:

$$R_a : \frac{a(x_{i_1}, \ldots, x_{i_k})}{D'_{i_1} = D_{i_1} \cap \Theta_{i_1} \ \cdots \ D'_{i_k} = D_{i_k} \cap \Theta_{i_k}} \ (P)$$

Dato un contraint store $\mathcal{C}$ l'approccio sarà quindi quello di:

- *semplificare* $\mathcal{C}$ in $\mathcal{CS} = \{a_1, \ldots, a_n\}$
- *applicare* per $i = 1, \ldots, n$ la corrispondente regola per a_i
- *propagare* il risultato della riduzione agli altri vincoli a_j, con $j \in \{1, \ldots, n\} \setminus \{i\}$.

Il *constraint solver* esegue questo processo in modo sequenziale: dopo aver esaminato il vincolo a_i propaga il risultato ai vincoli $a_{i+1}, \ldots, a_n$ per $i = 1, \ldots, n-1$.
Una volta esaminato l'ultimo vincolo a_n, il procedimento riparte dal primo vincolo a_1.

Questo processo viene iterato fino al raggiungimento di un **punto fisso**, cioè fino a quando ci si ritrova in uno dei seguenti casi:

(i) viene eseguita una scansione completa dei vincoli di $\mathcal{CS}$ senza effettuare alcuna riduzione di dominio: in questo caso il problema potrebbe essere risolto, fallito o nessuna delle precedenti.

(ii) uno dei domini delle variabili di $\mathcal{CS}$ diventa vuoto: in questo caso il $\mathcal{CS}$ è insoddisfacibile in quanto almeno uno dei suoi vincoli è contraddittorio: per il teorema 4.2.1 il problema è inconsistente.

Si può osservare come il raggiungimento di tale punto fisso sia garantito, in quanto:

- $\mathcal{CS}$ contiene un numero finito di vincoli atomici
- ogni vincolo di $\mathcal{CS}$ contiene un numero finito di variabili
- ogni variabile ha un dominio finito

- ogni regola di riduzione R_a restringe il dominio delle variabili che occorrono in a oppure lo lascia inalterato, senza aggiungere ulteriori vincoli.

Perciò, dopo un numero *finito* di iterazioni ci si ritroverà nel caso **(i)** oppure nel caso **(ii)**.

4.3 Regole di riduzione dei domini

In questa sezione verranno presentate le regole di riduzione dei domini per ciascun vincolo atomico del constraint store $\mathcal{CS}$, ed a ognuno di essi verrà associata una corrispondente nozione di consistenza locale.

4.3.1 Vincoli di dominio e uguaglianza

Regola 4.11 (Dominio)**.** Siano $x \in \mathcal{V}_{\mathcal{FD}}$ e $M \in \mathbb{M}_\alpha$. La **regola di domino** è definita nel seguente modo:

$$R_{::} : \frac{x :: I}{D'_x = D_x \cap M}$$

Proposizione 4.3.1. *La regola $R_{::}$ rispetta la nozione di hyper-arc consistency.*

Dimostrazione. Essendo coinvolta una sola variabile x, dimostro la *node consistency* di ::, cioè: $(\forall a \in D'_x)\ a \in M$.
Ma ciò segue dal fatto che $D'_x = D_x \cap M$ implica che $D'_x \subseteq M$ e quindi la tesi.

■

Già da questa regola è possibile intuire i vantaggi dell'utilizzo di multi-intervalli al posto degli intervalli.

Supponiamo infatti che una variabile v possa assumere solo due valori: 0 e 220. Se il suo dominio fosse $\mathbb{I}_\alpha$, allora sarebbe $v :: [0..220]$: in esso compaiono ben 119 valori inconsistenti.
Nel nostro caso invece questo problema viene risolto definendo il vincolo $v :: \{0, 220\}$.

Le potenzialità di questa maggiore precisione si noteranno meglio in seguito, quando si parlerà di *labeling*.

Regola 4.12 (Uguaglianza)**.** Sia $x \in \mathcal{V}_{\mathcal{FD}}$. Le **regole di uguaglianza** sono definite nel seguente modo:

$$R'_{=} : \frac{x = y}{D'_x = D_x \cap D_y \quad D'_y = D_x \cap D_y} \text{ (se } y \in \mathcal{V}_{\mathcal{FD}})$$

$$R''_{=} : \frac{x = k}{D'_x = D_x \cap \{k\}} \text{ (se } k \in \mathcal{Z}_{\mathcal{FD}})$$

Proposizione 4.3.2. *Le regole $R'_{=}$ e $R''_{=}$ rispettano la nozione di hyper-arc consistency.*

Dimostrazione. Partiamo da $R'_{=}$. Va dimostrato che $(\forall a \in D'_x)(\exists b \in D_y)\ a = b$ e $(\forall b \in D'_y)(\exists a \in D_x)\ a = b$ cioè che $D'_x \subseteq D_y$ e $D'_y \subseteq D_x$; ma ciò vale essendo $D'_x = D'_y = D_x \cap D_y \subseteq D_x, D_y$.
Per $R''_{=}$ bisogna provare invece che $(\forall a \in D'_x)\ a = k$ cioè $D'_x \subseteq \{k\}$: anche questo è immediato essendo $D'_x = D_x \cap \{k\} \subseteq \{k\}$.

■

Si può notare come l'applicazione di $R_{=}$ su $x = y$ equivalga all'applicazione di $R_{::}$ sui vincoli $x :: D_y$ e $y :: D_x$. Inoltre, se fosse $k \in \mathbb{Z}_\alpha$ allora l'applicazione di $R_{=}$ su $x = k$ equivarrebbe all'applicazione di $R_{::}$ su $x :: \{k\}$.

4.3.2 Vincoli di minore e minore o uguale

Regola 4.13 (Minore)**.** Sia $x \in \mathcal{V}_{\mathcal{FD}}$. Le **regole di minore** sono definite nel seguente modo:

$$R'_{<} : \frac{x < y}{D'_x = D_x \cap [-\alpha..D_y^+ - 1] \quad D'_y = D_y \cap [D_x^- + 1..\alpha]} \text{ (se } y \in \mathcal{V}_{\mathcal{FD}})$$

$$R''_{<} : \frac{x < k}{D'_x = D_x \cap [-\alpha..k - 1]} \text{ (se } k \in \mathcal{Z}_{\mathcal{FD}})$$

Proposizione 4.3.3. *Le regole $R'_{<}$ e $R''_{<}$ rispettano la nozione di hyper-arc consistency.*

Dimostrazione. Per $R'_{<}$ va dimostrato che $(\forall a \in D'_x)(\exists b \in D_y)\ a < b$. Sia allora $a \in D'_x$ generico, si ha che $-\alpha \leq a \leq D_y^+ - 1$ perciò $a < D_y^+ \in D_y$. In modo simmetrico si dimostra che $(\forall b \in D'_y)(\exists a \in D'_a)\ a < b$ provando che preso un generico $b \in D'_y$ si ha $D_x^- < b$.

Per $R''_{<}$ la dimostrazione è immediata: se $a \in D'_x$, allora sicuramente $a < k$ essendo in particolare $-\alpha \leq a \leq k - 1$.

■

Regola 4.14 (Minore o uguale)**.** Sia $x \in \mathcal{V}_{\mathcal{FD}}$. Le **regole di minore o uguale** sono definite nel seguente modo:

$$R'_{\leq} : \frac{x \leq y}{D'_x = D_x \cap [-\alpha..D_y^+] \quad D'_y = D_y \cap [D_x^-..\alpha]} \text{ (se } y \in \mathcal{V}_{\mathcal{FD}})$$

$$R''_{\leq} : \frac{x \leq k}{D'_x = D_x \cap [-\alpha..k]} \text{ (se } k \in \mathcal{Z}_{\mathcal{FD}})$$

Proposizione 4.3.4. *Le regole $R'_{\leq}$ e $R''_{\leq}$ rispettano la nozione di hyper-arc consistency.*

Dimostrazione. La dimostrazione è del tutto analoga a quella per le regole di minore. Per $R'_{\leq}$, si prova che preso un generico $a \in D'_x$ si ha $a \leq D_y^+$ e simmetricamente che preso un generico $b \in D'_y$ si ha $D_x^- \leq b$; per $R''_{\leq}$ è facile notare che per ogni $a \in D'_x$ si ha $a \leq k$.

■

4.3.3 Vincolo di diverso

Regola 4.15 (Diverso)**.** Sia $x \in \mathcal{V}_{\mathcal{FD}}$. Le **regole di diverso**[4] sono definite nel seguente modo:

$$R'_{\neq} : \frac{x \neq y}{D'_x = D_x \setminus \{k\} \quad D'_y = D_y} \text{ (se } y \in \mathcal{V}_{\mathcal{FD}} \text{ e } D_y = \{k\})$$

$$R''_{\neq} : \frac{x \neq y}{D'_x = D_x \quad D'_y = D_y \setminus \{k\}} \text{ (se } x \in \mathcal{V}_{\mathcal{FD}} \text{ e } D_x = \{k\})$$

$$R'''_{\neq} : \frac{x \neq k}{D'_x = D_x \setminus \{k\}} \text{ (se } k \in \mathcal{Z}_{\mathcal{FD}})$$

Proposizione 4.3.5. *Le regole $R'_{\neq}$, $R''_{\neq}$ e $R'''_{\neq}$ rispettano la nozione di hyper-arc consistency.*

Dimostrazione. Dimostriamo la prima regola. Sia $a \in D'_x$ generico, allora preso $b \in D_y = \{k\}$ si ha che $b = k$; dalla definizione di D'_x segue $a \neq b$. Analogamente, se $b \in D'_y$ allora $b = k$; per definizione di D'_x preso un qualsiasi $a \in D'_x$ si ha che $a \neq b$.

[4]Si noti che per queste regole viene utilizzata la differenza insiemistica anziché l'intersezione; tuttavia, questa condizione non è restrittiva: per ogni $A, B, C \subseteq \mathbb{Z}$ si ha infatti che $A = B \setminus C \iff A = B \cap {\sim}C$

La dimostrazione delle altre due regole è del tutto simile.

■

E' importante osservare come una tale definizione non sarebbe significativa nel dominio degli intervalli, o quantomeno risulterebbe limitata ai soli casi 'degeneri'.

Prendiamo ad esempio un $CSP(\mathcal{FD})$ $\mathcal{P} = \langle\{x\}, D_x, \{x :: [a..b], x \neq k\}\rangle$ in cui $D_x \in \mathbb{I}_\alpha$ e $a, b, k \in \mathbb{Z}_\alpha$ tali che $a < k < b$.
In questo caso, non è possibile rimuovere l'elemento k dall'intervallo $[a..b]$: l'insieme $[a..k-1] \cup [k+1..b]$ risultante non è un intervallo.
L'applicazione di $\setminus$ esteso a $\mathbb{I}_\alpha$ risulta infatti:

$$[a..b] \setminus_{\mathbb{I}} \{k\} = \mathcal{CH}_\alpha([a..b] \setminus \{k\}) = \mathcal{CH}_\alpha([a..k-1] \cup [k+1..b]) = [a..b]$$

Quindi, i possibili approcci alla risoluzione possono essere:

- lasciare inalterato il dominio di x
- scomporre il problema $\mathcal{P}$ in due sotto-problemi $\mathcal{P}_1$ e $\mathcal{P}_2$ con $\mathcal{P}_1 = \langle\{x\}, D_x, \{x :: [a..k-1]\}\rangle$ e $\mathcal{P}_2 = \langle\{x\}, D_x, \{x :: [k+1..b]\}\rangle$.

Nel primo caso, nel dominio di x viene mantenuto il valore inconsistente k.
Nel secondo invece ci si riconduce ad un problema di ricerca della soluzione *non-deterministico* (apro due 'strade' per risolvere $\mathcal{P}_1$ e $\mathcal{P}_2$) provocando una non trascurabile perdita di efficienza.

Utilizzando invece i multi-intervalli, la regola $R'''_{\neq}$ comporta la riduzione del dominio di x al multi-intervallo $\{a..k-1, k+1..b\}$: il valore inconsistente k viene rimosso in modo *deterministico*.

I vantaggi di questa soluzione sono quindi evidenti: ottengo maggiore precisione ad un costo computazionale inferiore.

4.3.4 Vincolo di somma

Regola 4.16 (Somma)**.** Siano $x, z \in \mathcal{V}_{\mathcal{FD}}$. Le **regola di somma** sono definite nel seguente modo:

$$R'_+ : \frac{z = x + y}{\begin{array}{l} D'_x = D_x \cap (\mathcal{CH}_\alpha(D_z) \ominus \mathcal{CH}_\alpha(D_y)) \\ D'_y = D_y \cap (\mathcal{CH}_\alpha(D_z) \ominus \mathcal{CH}_\alpha(D_x)) \\ D'_z = D_z \cap (\mathcal{CH}_\alpha(D_x) \oplus \mathcal{CH}_\alpha(D_y)) \end{array}} \text{ (se } y \in \mathcal{V}_{\mathcal{FD}})$$

$$R''_+ : \frac{z = x + k}{\begin{array}{l} D'_x = D_x \cap (D_z \boxminus \{k\}) \\ D'_z = D_z \cap (D_x \boxplus \{k\}) \end{array}} \text{ (se } k \in \mathcal{Z}_{\mathcal{FD}})$$

Proposizione 4.3.6. *La regola R'_+ non rispetta la nozione di hyper-arc consistency.*

Dimostrazione. Si prendano ad esempio $x, y, z \in \mathcal{V}_{\mathcal{FD}}$ con $D_x = [-20..20]$, $D_y = [1..4]$ e $D_z = \{1..2, 15..18\}$. La regola R'_+ determina una riduzione dei domini tale che:

- $D'_x = D_x \cap ([1..18] \ominus [1..4]) = [-20..20] \cap [-3..17] = [-3..17]$
- $D'_y = D_y \cap ([1..18] \ominus [-20..20]) = [1..4] \cap [-19..38] = [1..4]$
- $D'_z = D_z \cap ([-20..20] \oplus [1..4]) = \{1..2, 15..18\} \cap [-19..24] = \{1..2, 15..18\}$

Ora, preso $a = 5 \in D'_x$ si ha che non esistono $b \in D_y$ e $c \in D_z$ tali che $c = a + b$ quindi con la regola R'_+ si ha che $+$ non è hyper-arc consistency.

■

Proposizione 4.3.7. *La regola R'_+ rispetta la nozione di range consistency.*

Dimostrazione. Per la definizione di range consistency, va provato che:

- $(\forall a \in D'_x)(\exists b \in \mathcal{CH}_\alpha(D_y))(\exists c \in \mathcal{CH}_\alpha(D_z))\ c = a + b$
- $(\forall b \in D'_y)(\exists a \in \mathcal{CH}_\alpha(D_x))(\exists c \in \mathcal{CH}_\alpha(D_z))\ c = a + b$
- $(\forall c \in D'_z)(\exists a \in \mathcal{CH}_\alpha(D_x))(\exists b \in \mathcal{CH}_\alpha(D_y))\ c = a + b$

Dimostriamo solo la prima formula, la dimostrazione delle altre è identica. Si noti anzitutto che $D'_x \subseteq (\mathcal{CH}_\alpha(D_z) \ominus \mathcal{CH}_\alpha(D_y)) \subseteq \mathcal{CH}_\alpha(D_z) - \mathcal{CH}_\alpha(D_y)$. Quindi, preso un generico $a \in D'_x$ si ha che $a \in \mathcal{CH}_\alpha(D_z) - \mathcal{CH}_\alpha(D_y)$. Allora, per la definizione di sottrazione insiemistica data in 3.6 esistono $c \in \mathcal{CH}_\alpha(D_z)$ e $b \in \mathcal{CH}_\alpha(D_y)$ tali che $a = c - b$ cioè $c = a + b$.

■

Proposizione 4.3.8. *La regola R''_+ rispetta la nozione di hyper-arc consistency.*

Dimostrazione. Per la definizione di range consistency, va provato che:

- $(\forall a \in D'_x)(\exists c \in D_z)\ c = a + k$
- $(\forall c \in D'_z)(\exists a \in D_x)\ c = a + k$

Dimostriamo la prima formula. Sia $a \in D'_x$; allora $a \in D_z \boxminus \{k\}$ da cui segue che esiste $c \in D_z$ tale che $a = c - k$ perciò $c = a + k$. Analogamente si dimostra l'altra formula.

■

A questo punto è lecito chiedersi perché si è scelto di sovra-approssimare la somma in R'_+, rilassando l'hyper-arc consistency nella range consistency. La risposta consiste ovviamente nella maggiore efficienza di quest'ultima nozione.

Se infatti R'_+ fosse stata definita come:

$$\mathbf{R}'_+ : \frac{z = x + y}{D'_x = D_x \cap (D_z \boxminus D_y) \quad D'_y = D_y \cap (D_z \boxminus D_x) \quad D'_z = D_z \cap (D_x \boxplus D_z)}$$

Allora rispetterebbe la hyper-arc consistency.
Ad esempio, se $x, y, z \in \mathcal{V}_{\mathcal{FD}}$ con $D_x = [-20..20]$, $D_y = [1..4]$ e $D_z = \{1..2, 15..18\}$ l'applicazione di $\mathbf{R}'_+$ comporterebbe una riduzione dei domini tale che $D'_x = \{-3..1, 11..17\}$ anziché l'insieme $[-3..17]$ risultante dall'applicazione di R'_+.

Tuttavia, il maggiore costo computazionale del calcolo di $\mathbf{R}'_+$ rispetto ad R'_+ fa preferire quest'ultima regola.

Si noti infine che se D_x, D_y, D_z sono intervalli allora R'_+ rispetta la nozione di hyper-arc consistency.

4.3.5 Vincolo di moltiplicazione

Regola 4.17 (Moltiplicazione)**.** Siano $x, z \in \mathcal{V}_{\mathcal{FD}}$. Le **regole di moltiplicazione** sono definite nel seguente modo:

$$R'_* : \frac{z = x \cdot y}{\begin{array}{l} D'_x = D_x \cap (\mathcal{CH}_\alpha(D_z) \oslash \mathcal{CH}_\alpha(D_y)) \\ D'_y = D_y \cap (\mathcal{CH}_\alpha(D_z) \oslash \mathcal{CH}_\alpha(D_x)) \\ D'_z = D_z \cap (\mathcal{CH}_\alpha(D_x) \odot \mathcal{CH}_\alpha(D_y)) \end{array}} \text{ (se } y \in \mathcal{V}_{\mathcal{FD}})$$

$$R''_* : \frac{z = x \cdot k}{\begin{array}{l} D'_x = D_x \cap (D_z \div \{k\}) \\ D'_z = D_z \cap (D'_x \bullet \{k\}) \end{array}} \text{ (se } k \in \mathcal{Z}_{\mathcal{FD}})$$

dove, per ogni multi-intervallo $I = I_1 \cup \ldots \cup I_n$ e $J = J_1 \cup \ldots \cup J_m$:

$$I \bullet J = \bigcup \{I_h \odot J_k : 1 \leq h \leq n \text{ e } 1 \leq k \leq m\}$$
$$I \div J = \bigcup \{I_h \oslash J_k : 1 \leq h \leq n \text{ e } 1 \leq k \leq m\}$$

Proposizione 4.3.9. *La regola R'_* non rispetta le nozioni di hyper-arc consistency, range consistency e bound consistency.*

Dimostrazione. Si prendano ad esempio $x, y, z \in \mathcal{V}_{\mathcal{FD}}$ con $D_x = [-2..1]$, $D_y = [-3..10]$ e $D_z = [8..10]$. La regola R'_* non determina alcuna riduzione dei domini, infatti:

- $D'_x = D_x \cap ([8..10] \oslash [-3..10]) = [-2..1] \cap [-10..10] = [-2..1]$
- $D'_y = D_y \cap ([8..10] \oslash [-2..1]) = [-3..10] \cap [-10..10] = [-3..10]$
- $D'_z = D_z \cap ([-2..1] \odot [-3..10]) = [8..10] \cap [-20..10] = [8..10]$

Ora, preso $b = D'^{-}_y = -3$ si ha che non esistono $a \in \mathcal{CH}_\alpha(D_x)$ e $c \in \mathcal{CH}_\alpha(D_z)$ tali che $c = a \cdot b$ quindi con la regola R'_* si ha che $\cdot$ non è bound consistency. Per contronominale, dal teorema 2.2.1 segue che:
$\mathcal{P}$ non è bound cons. $\implies$ $\mathcal{P}$ non è range cons. $\implies$ $\mathcal{P}$ non è hyper-arc cons. Perciò, R'_* non rispetta nemmeno le nozioni di range consistency e hyper-arc consistency.

■

Proposizione 4.3.10. *La regola R''_* non rispetta le nozioni di hyper-arc consistency e range consistency.*

Dimostrazione. Per quanto visto, basta dimostrare che R''_* non rispetta la range consistency. Siano ad esempio $x, z \in \mathcal{V}_{\mathcal{FD}}$ e $k \in \mathcal{Z}_{\mathcal{FD}}$ con $D_x = [0..1]$, $D_z = [-11..11]$ e $k = \{2\}$. La regola R''_* determina una riduzione dei domini tale che:

- $D'_x = D_x \cap ([-11..11] \div \{2\}) = [0..1] \cap [-5..5] = [0..1]$
- $D'_z = D_z \cap ([0..1] \bullet \{2\}) = [-11..11] \cap [0..2] = [0..2]$

Ora, preso $c = 1 \in D'_z$ si ha che non esiste $a \in \mathcal{CH}_\alpha(D_x) = [0..1]$ tale che $a \cdot 2 = 1$ quindi con la regola R''_* si ha che $\cdot$ non è range consistency.

■

Proposizione 4.3.11. *La regola R''_* rispetta la nozione di bound consistency.*

Dimostrazione. Per definizione di bound consistency, va dimostrato che:

- $(\exists\ a, b \in \mathcal{CH}_\alpha(D_x))\ D'^{-}_z = a \cdot k$ e $D'^{+}_z = b \cdot k$
- $(\exists\ c, d \in \mathcal{CH}_\alpha(D_z))\ c = D'^{-}_x \cdot k$ e $d = D'^{+}_x \cdot k$

Dimostriamo solo una parte dell'asserto, il resto è del tutto simile.
Supponiamo $k > 0$ (la dimostrazione per $k \leq 0$ è simmetrica) e prendiamo $a = D'^{-}_x \in \mathcal{CH}_\alpha(D_x)$. Supponiamo ora per assurdo che $D'^{-}_z \neq D'^{-}_x \cdot k$; allora deve essere $D^{-}_z > D'^{-}_x \cdot k$, altrimenti risulterebbe $D'^{-}_z = max(D^{-}_z, D'^{-}_x \cdot k) = D'^{-}_x \cdot k$ contro l'ipotesi fatta. Quindi, $D'^{-}_x < D^{-}_z/k$ il che è altrettanto assurdo essendo $D'^{-}_x = max(D^{-}_x, D^{-}_z/k)$. Perciò, dev'essere $D'^{-}_z = D'^{-}_x \cdot k$.
Analogamente si dimostra valere l'asserto con $b = D'^{+}_x$, $c = D'^{-}_z$ e $d = D'^{+}_z$.

■

Dalla proposizione segue le regole di moltiplicazione effettuano una forte sovra-approssimazione, addirittura maggiore di quella vista per le regole di somma.
In effetti, il problema di una caratterizzazione ottimale della moltiplicazione non è banale. Ad esempio, definire regole del tipo:

$$\mathbf{R}'_* : \frac{z = x \cdot y}{D'_x = D_x \cap (D_z \boxslash D_y) \quad D'_y = D_y \cap (D_z \boxslash D_x) \quad D'_z = D_z \cap (D_x \boxdot D_z)}$$

$$\mathbf{R}''_* : \frac{z = x \cdot k}{D'_x = D_x \cap (D_z \boxslash \{k\}) \quad D'_z = D_z \cap (D_x \boxdot \{k\})}$$

che mantengano la hyper-arc consistency risulterebbe decisamente troppo costoso dal punto di vista computazionale.
Si è quindi deciso di mantenere l'approccio descritto in [2], preferendo l'efficienza della risoluzione rispetto alla sua precisione.

Si noti comunque che è comunque possibile definire altre regole '*ad hoc*' che permettano di aumentare il potere riduttivo in alcuni casi particolari.
Si prenda ad esempio il vincolo $y = x \cdot x$ con $D_x = [-10..10]$ e $D_y = \{9\}$.
La regola R'_* effettuerà una riduzione di D_x tale che:

$$D'_x = D_x \cup (\{9\} \oslash [-10..10]) = [-10..10] \cap [-9..9] = [-9..9]$$

effettuando una sovra-approssimazione dell'insieme $\{-3, 3\}$ dei possibili valori che $x = \sqrt{y}$ può assumere.

Per gestire casi particolari di questo tipo, può risultare quindi comodo aggiungere una nuova regola R'''_* del tipo:

$$R'''_* : \frac{z = x \cdot y}{\begin{array}{l} D'_x = D_x \cap \{-\sqrt{k}, \sqrt{k}\} \\ D'_y = D_y \cap \{-\sqrt{k}, \sqrt{k}\} \\ D'_z = D_z \end{array}} \text{ (se } x = y \text{ e } D'_z = \{k\})$$

Con questa regola, nell'esempio precedente risulterà:

$$D'_x = [-10..10] \cap \{-3, 3\} = \{-3, 3\}$$

4.4 Consistenza globale e ricerca della soluzione

Come già accennato nel capitolo 2 la consistenza locale, ottenuta mediante riduzione dei domini e propagazione dei vincoli, non è sempre sufficiente per garantire la consistenza globale di un CSP.

Riprendendo l'esempio 1 ci si accorge come le regole $R_{\neq}$ non comportino alcuna riduzione di dominio, per cui la propagazione raggiunge un punto fisso senza individuare l'insoddisfacibilità del constraint store.

Inoltre, un CSP consistente spesso non è risolto: si pensi ad esempio a $\mathcal{P} = \langle \{x, y\}, D_x \times D_y, \{x < y\} \rangle$ con $D_x = \{0\}$ e $D_y = \{1, 2\}$.
In questo caso, le due soluzioni $d_1 = \langle 0, 1 \rangle$ e $d_2 = \langle 0, 2 \rangle$ non vengono individuate.

Per superare l'incompletezza del solver servono quindi apposite tecniche di *consistenza globale* e *ricerca della(e) soluzione(i)*; tra i possibili approcci noi utilizzeremo quello del *labeling*.

4.4.1 Labeling

In generale, data una n-upla di variabili $V = \langle x_1, \ldots, x_n \rangle \in {\mathcal{V}_{\mathcal{FD}}}^n$, fare **labeling su** V significa assegnare ad ogni x_i un valore del suo dominio.

È evidente che considerando ogni possibile labeling su tutte le variabili del constraint store ottengo la *completezza* della risoluzione (considero ogni possibile funzione di assegnazione).

Tuttavia, la brutalità di questo metodo implica tempi di calcolo non ragionevoli.

Per questa ragione, la ricerca della soluzione mediante labeling viene migliorata attraverso apposite *euristiche*.

Definizione 4.18 (Euristiche di labeling)**.** Sia $k \in \mathbb{N}$ generico.

Una **Variable Choice Heuristic** è una funzione $\rho : {\mathcal{V}_{\mathcal{FD}}}^k \longrightarrow \mathcal{V}_{\mathcal{FD}}$ tale che:

$$\rho(x_1, \ldots, x_k) \in \{x_1, \ldots, x_k\} \quad \text{per ogni } \langle x_1, \ldots, x_k \rangle \in {\mathcal{V}_{\mathcal{FD}}}^k$$

Una **Value Choice Heuristic** è una funzione $\lambda : \mathcal{V}_{\mathcal{FD}} \longrightarrow \mathbb{Z}_\alpha$ tale che:

$$\lambda(x) \in D_x \quad \text{per ogni } x \in \mathcal{V}_{\mathcal{FD}}$$

Dalla definizione segue che esistono un infinità di euristiche per la scelta di variabili e valori.
Vediamo quindi alcuni esempi.

Esempio 3 (Variable Choice Heuristic). Sia $V = \langle x_1, \ldots, x_n \rangle \in {\mathcal{V}_{\mathcal{FD}}}^n$.
Possibili euristiche ρ per la scelta di una variabile $x \in \{x_1, \ldots, x_n\}$ sono:

left-most:	$\rho(V) = x_1$	
mid-most:	$\rho(V) = x_k,$	dove $k = \left\lfloor \frac{n}{2} \right\rfloor$
right-most:	$\rho(V) = x_n$	
min:	$\rho(V) = x_k,$	dove $D_k^- = min_{\leq}\{D_i^- : 1 \leq i \leq n\}$
max:	$\rho(V) = x_k,$	dove $D_k^+ = max_{\leq}\{D_i^+ : 1 \leq i \leq n\}$
first-fail:	$\rho(V) = x_k,$	dove $D_k = min_{\#}\{D_i : 1 \leq i \leq n\}$[5]
random:	$\rho(V) = x_k,$	dove $k = random([1..n])$

Intuitivamente, le euristiche *left-most*, *mid-most* e *right-most* selezionano rispettivamente le variabili che stanno in testa, in mezzo ed in coda a V.
Min e *max* scelgono la variabile il cui dominio ha il minor GLB ed il maggior LUB rispettivamente; *first-fail* seleziona la variabile con dominio più piccolo (nel caso più variabili soddisfino questi criteri, la scelta è arbitraria; ad esempio si può prendere la variabile *left-most* tra quelle candidate).
Random estrae una variabile in modo casuale ed *equiprobabile* fra $\{x_1, \ldots, x_n\}$.

Esempio 4 (Value Choice Heuristic). Siano $x \in \mathcal{V}_{\mathcal{FD}}$ e $D_x = I_1 \cup \ldots \cup I_n$.

[5] Presa una collezione di insiemi $\mathcal{X} \subseteq \mathcal{P}(\mathbb{Z})$, con la notazione $min_{\#}(\mathcal{X})$ (rispettivamente, $max_{\#}(\mathcal{X})$) verrà indicato l'insieme $A \in \mathcal{X}$ con la cardinalità minima (massima).
Dato un insieme $A \subseteq \mathbb{Z}$, $random(A)$ restituisce un valore $x \in A$ *casuale ed uniformemente distribuito.*

Possibili euristiche λ per la scelta di un valore $d \in D_x$ sono:

$$\begin{aligned}
\textbf{glb:} \quad & \lambda(x) = D_x^- \\
\textbf{lub:} \quad & \lambda(x) = D_x^+ \\
\textbf{mid-most:} \quad & \lambda(x) = \left\lfloor \frac{I_k^- + I_k^+}{2} \right\rfloor, && \text{dove } k = \left\lfloor \frac{n}{2} \right\rfloor \\
\textbf{median:} \quad & \lambda(x) = Me(D_x)^6 \\
\textbf{equi-random:} \quad & \lambda(x) = d, && \text{dove } d = random(D_x) \\
\textbf{range-random:} \quad & \lambda(x) = random([I_k^- .. I_k^+]), && \text{dove } k = random([1..n]) \\
\textbf{mid-random:} \quad & \lambda(x) = \left\lfloor \frac{I_k^- + I_k^+}{2} \right\rfloor, && \text{dove } k = random([1..n])
\end{aligned}$$

Le euristiche *glb* e *lub* selezionano rispettivamente il GLB ed il LUB del dominio di x.
Mid-most sceglie il punto medio dell'intervallo 'centrale' di D_x.
Equi-random estrae un valore casuale in D_x mentre *Range-random* estrae dapprima un intervallo $I_k \subseteq D_x$, quindi seleziona un valore casuale in I_k.

Si osservi che quest'ultima euristica non conserva l'equi-distribuzione: la probabilità che venga estratto un valore $d \in D_x$ è *inversamente proporzionale* all'ampiezza dell'intervallo I_k a cui appartiene d.
Formalmente, detta $\mathbb{P}[E]$ la probabilità che si verifichi un dato evento E, si ha che per ogni $d \in D_x$:

$$\mathbb{P}[\lambda(x) = d] = \begin{cases} \dfrac{1}{|D_x|}, & \text{se } \lambda \text{ è equi-random} \\ \dfrac{1}{n} \sum_{j=1}^{n} \dfrac{\chi_{I_j}(d)}{|I_j|} & \text{se } \lambda \text{ è range-random}^7 \end{cases}$$

Si noti che se $D_x \in \mathbb{I}_\alpha$ allora equi-random e range-random sono di fatto la stessa euristica.

Mid-random è una soluzione ibrida: dapprima viene scelto un intervallo I_k in modo casuale e quindi viene selezionato il suo punto medio.
Ogni punto medio ha quindi una probabilità pari a $\dfrac{1}{n}$ di essere estratto.

[6] Dato un insieme $A \subseteq \mathbb{Z}$, con $Me(A)$ indichiamo la *mediana* dell'insieme di valori A. Si noti che se $|A|$ è pari, considereremo $Me(A)$ come il *minimo* fra i due valori mediani dell'insieme A.

[7] Con χ_A è indichiamo la *funzione caratteristica* di A, tale che $\chi_A(x) = \begin{cases} 0, & \text{se } x \notin A \\ 1 & \text{se } x \in A \end{cases}$

Si noti che se $D_x \in \mathbb{I}_\alpha$ allora mid-random, mid-most e median sono di fatto la stessa euristica.

Esempio 5. Sia $V = \langle x, y, z \rangle$ con $D_x = \{1..10, 28..30\}$, $D_y = \{1..50, 100, 1000\}$ e $D_z = [0..40]$. Vediamo come opera ρ a seconda del tipo di euristica scelto:

$$\begin{aligned}
\text{left-most:} &\quad \rho(V) = x \\
\text{mid-most:} &\quad \rho(V) = y \\
\text{right-most:} &\quad \rho(V) = z \\
\text{min:} &\quad \rho(V) = z \\
\text{max:} &\quad \rho(V) = y \\
\text{first-fail:} &\quad \rho(V) = x \\
\text{random:} &\quad \mathbb{P}[\rho(V) = x] = \mathbb{P}[\rho(V) = y] = \mathbb{P}[\rho(V) = z] = \frac{1}{3}
\end{aligned}$$

Analogamente, osserviamo come si comporta λ sulle variabili di V:

$$\begin{array}{rlll}
\text{glb:} & \lambda(x) = 1 & \lambda(y) = 1 & \lambda(z) = 0 \\
\text{lub:} & \lambda(x) = 30 & \lambda(y) = 1000 & \lambda(z) = 40 \\
\text{mid-most:} & \lambda(x) = 5 & \lambda(y) = 100 & \lambda(z) = 20 \\
\text{median:} & \lambda(x) = 7 & \lambda(y) = 26 & \lambda(z) = 20
\end{array}$$

Inoltre, per ogni $a \in D_x$, $b \in D_y$ e $c \in D_z$ si ha che:

$$\begin{aligned}
\text{equi-random:} &\quad \mathbb{P}[\lambda(x) = a] = \frac{1}{13} \\
&\quad \mathbb{P}[\lambda(y) = b] = \frac{1}{52} \\
&\quad \mathbb{P}[\lambda(z) = c] = \frac{1}{41}
\end{aligned}$$

$$\text{range-random:} \quad \mathbb{P}[\lambda(x) = a] = \begin{cases} \frac{1}{20}, & \text{se } a \in [1..10] \\ \frac{1}{6} & \text{se } a \in [28..30] \end{cases}$$

$$\mathbb{P}[\lambda(y) = b] = \begin{cases} \frac{1}{150}, & \text{se } b \in [1..50] \\ \frac{1}{3}, & \text{se } b = 100 \\ \frac{1}{3} & \text{se } b = 1000 \end{cases}$$

$$\mathbb{P}[\lambda(z) = c] = \frac{1}{41}$$

$$\text{mid-random:} \quad \mathbb{P}[\lambda(x) = a] = \begin{cases} \frac{1}{2}, & \text{se } a \in \{5, 29\} \\ 0 & \text{altrimenti} \end{cases}$$

$$\mathbb{P}[\lambda(y) = b] = \begin{cases} \frac{1}{3}, & \text{se } b \in \{25, 100, 1000\} \\ 0 & \text{altrimenti} \end{cases}$$

$$\lambda(z) = 20$$

Ovviamente, le euristiche che fanno uso della funzione *random* non si comportano in modo 'deterministico': applicando la stessa euristica al medesimo argomento potrei ottenere risultati differenti.

4.4.2 Ricerca della soluzione

Dopo aver definito la nozione di labeling e le relative euristiche, occupiamoci ora del processo di risoluzione vero e proprio.

Definizione 4.19 (Operatore di riduzione)**.** Sia $\mathcal{P}$ un $CSP(\mathcal{FD})$. Definisco **operatore di riduzione** l'operatore μ tale che $\mu(\mathcal{P})$ è il CSP ottenuto da $\mathcal{P}$ applicando le regole di riduzione dei domini e la propagazione dei vincoli al constraint store semplificato di $\mathcal{P}$, fino al raggiungimento di un punto fisso.

Osservazione 8 (Operatore di riduzione). É immediato osservare che:

- μ *riduce* i domini di $\mathcal{P}$, cioè $\mu(\mathcal{P}) \preceq \mathcal{P}$
- μ *conserva* le soluzioni di $\mathcal{P}$, cioè $\mu(\mathcal{P}) \equiv \mathcal{P}$
- μ è *idempotente*: $(\mu \circ \mu)(\mathcal{P}) = \mu(\mathcal{P})$.

L'approccio generale di ricerca della soluzione è il seguente:

1. il CSP orginale $\mathcal{P}$ viene ridotto in $\mathcal{P}' = \mu(\mathcal{P})$
2. se $\mathcal{P}'$ è risolto oppure fallito ci si ferma; in caso contrario $\mathcal{P}'$ viene decomposto in $k > 1$ sotto-problemi $\mathcal{P}'_1, \ldots, \mathcal{P}'_k$ tali che:
   - $\mathcal{P}'_i \prec \mathcal{P}'$ per $i = 1, \ldots, k$
   - $\bigcup_{i=1}^{k} \xi(\mathcal{P}'_i) = \xi(\mathcal{P}')$
3. ogni $\mathcal{P}'_i$, secondo un certo ordine, viene ridotto e ulteriormente scomposto: il processo riparte ricorsivamente da 2. fino ad ottenere un CSP risolto o fallito.

Facciamo alcune considerazioni. Anzitutto, le condizioni poste al punto 2. garantiscono la terminazione dell'algoritmo (ad ogni passo ottengo CSP strettamente minori) e la conservazione delle soluzioni del CSP originale.

Inoltre, tale processo di ricerca può essere modellato da un **albero** nel quale:

- ogni *nodo* dell'albero corrisponde a un CSP
- la *radice* corrisponde al CSP originale $\mathcal{P}$
- se un CSP P viene suddiviso in $P_1, \ldots, P_k$ allora il nodo corrispondente a P avrà esattamente k *figli* corrispondenti a $P_1, \ldots, P_k$
- le *foglie* corrispondono a CSP risolti o falliti.

Quindi, la ricerca della soluzione si riduce ad un problema di ricerca su un tale albero, secondo una certa *strategia*.

É importante notare che la scelta di quanti e quali sotto-problemi considerare dipenderà da una o più euristiche.
Analogamente, anche la scelta dell'*ordine* di risoluzione dei sotto-problemi è arbitraria.

Nel nostro caso, ci si affiderà alle euristiche di labeling; esistono tuttavia tecniche alternative quali ad esempio la *bisezione*[2].

Sia quindi $\mathcal{P}$ un $CSP(\mathcal{FD})$ e V una n-upla di variabili di $\mathcal{P}$ sulle quali fare labeling, mediante apposite euristiche ρ e λ.
La ricerca avviene nel seguente modo:

1. il CSP orginale $\mathcal{P}$ viene ridotto in $\mathcal{P}' = \mu(\mathcal{P})$
2. se $\mathcal{P}'$ è risolto oppure fallito ci si ferma
3. se $\mathcal{P}'$ non è nè risolto nè fallito:
   - si seleziona una *variabile* $x = \rho(V)$ con $|D_x| > 1$ che viene poi rimossa da V
   - si seleziona un *valore* $d = \lambda(x)$
   - si definiscono due intervalli $L_x, U_x \subset D_x$ nel seguente modo:
     $$L_x = min_{\#}([D_x^- .. d-1], [d+1 .. D_x^+])$$
     $$U_x = max_{\#}([D_x^- .. d-1], [d+1 .. D_x^+])$$
   - suddivido $\mathcal{P}'$ in *al più* tre sottoproblemi:
     (i) $\mathcal{P}'_d$, ottenuto da $\mathcal{P}'$ aggiungendo il vincolo $x = d$
     (ii) $\mathcal{P}'_L$, ottenuto da $\mathcal{P}'$ aggiungendo il vincolo di dominio $x :: L_x$, nel caso sia $L_x \neq \varnothing$
     (iii) $\mathcal{P}'_U$, ottenuto da $\mathcal{P}'$ aggiungendo il vincolo di dominio $x :: U_x$
4. dapprima risolvo ricorsivamente $\mu(\mathcal{P}'_d)$, quindi (nel caso $L_x \neq \varnothing$) $\mu(\mathcal{P}'_L)$ e infine $\mu(\mathcal{P}'_U)$.

Una risoluzione di questo tipo è evidentemente meno efficiente rispetto al processo di riduzione dei domini e propagazione dei vincoli.

Infatti, la scomposizione in sotto-problemi causa la generazione di *choice-points* nei nodi interni dell'albero di ricerca, per tenere traccia dei problemi non ancora risolti.
Una volta giunti ad un nodo foglia, la computazione riprende dall'ultimo choice-point creato (*backtracking cronologico*).

Tuttavia, nel caso ci interessi conoscere una sola soluzione, è possibile interrompere il processo al primo nodo foglia che corrisponda ad un CSP risolto.

Inoltre, ad ogni nodo vengono creati al più due choice-points: uno relativo a $\mathcal{P}'_U$ ed eventualmente un altro relativo a $\mathcal{P}'_L$.

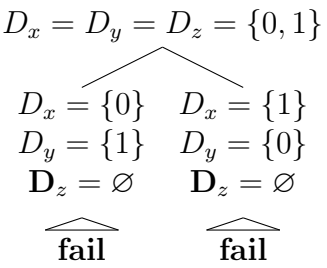
Si noti infine che la completezza della risoluzione dipende dalle variabili che occorrono in V: può essere buona norma minimizzare il numero delle variabili sulle quali fare labeling per ridurre il costo computazionale.

Concludiamo il capitolo illustrando alcuni esempi di risoluzione.

Esempio 6 (Risoluzione con labeling). Si consideri il CSP dell'esempio 1:

$$\mathcal{P} = \langle \{x, y, z\}, \{0, 1\} \times \{0, 1\} \times \{0, 1\}, \{x \neq y, y \neq z, z \neq x\} \rangle$$

In questo caso la riduzione non porta alcun beneficio: $\mu(\mathcal{P}) = \mathcal{P}$.
Vediamo quindi come attraverso il labeling sia possibile individuare l'inconsistenza di $\mathcal{P}$.
Come detto in precedenza, non è sempre necessario considerare tutte le variabili di $\mathcal{P}$ per risolvere il problema; prendiamo ad esempio $V = \langle x \rangle$ e λ mid-most (in questo caso, essendo V costituito da una sola variabile, la scelta di ρ è ininfluente).
La risoluzione è descritta dal seguente albero di ricerca, nel quale in ogni nodo vengono indicati i domini del corrispondente sotto-problema dopo il processo di riduzione:

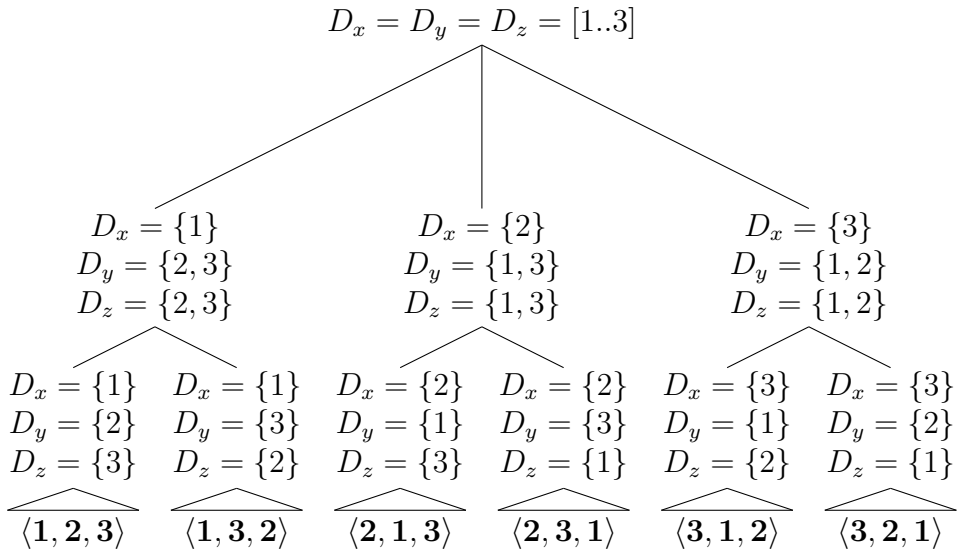


Come si può notare, ogni foglia dell'albero di ricerca corrisponde ad un CSP fallito: non esiste alcuna soluzione, quindi $\mathcal{P}$ è inconsistente.

Si consideri ora il seguente $CSP(\mathcal{FD})$:

$$\mathcal{P} = \langle \{x, y, z\}, [1..3] \times [1..3] \times [1..3], \{x \neq y, y \neq z, z \neq x\} \rangle$$

Anche in questo caso, $\mu(\mathcal{P}) = \mathcal{P}$. Tuttavia, facendo labeling su due delle tre variabili di $\mathcal{P}$ riesco ad ottenere tutte le *permutazioni* dell'insieme $\{1, 2, 3\}$.
Prendiamo quindi $V = \langle x, y \rangle$, ρ left-most e λ glb.
L'albero di ricerca risultante è il seguente:


$D_x = D_y = D_z = [1..3]$
$D_x = \{1\}$ $D_y = \{2,3\}$ $D_z = \{2,3\}$
$D_x = \{2\}$ $D_y = \{1,3\}$ $D_z = \{1,3\}$
$D_x = \{3\}$ $D_y = \{1,2\}$ $D_z = \{1,2\}$
$D_x = \{1\}$ $D_y = \{2\}$ $D_z = \{3\}$
$D_x = \{1\}$ $D_y = \{3\}$ $D_z = \{2\}$
$D_x = \{2\}$ $D_y = \{1\}$ $D_z = \{3\}$
$D_x = \{2\}$ $D_y = \{3\}$ $D_z = \{1\}$
$D_x = \{3\}$ $D_y = \{1\}$ $D_z = \{2\}$
$D_x = \{3\}$ $D_y = \{2\}$ $D_z = \{1\}$
$\langle \mathbf{1}, \mathbf{2}, \mathbf{3} \rangle$
$\langle \mathbf{1}, \mathbf{3}, \mathbf{2} \rangle$
$\langle \mathbf{2}, \mathbf{1}, \mathbf{3} \rangle$
$\langle \mathbf{2}, \mathbf{3}, \mathbf{1} \rangle$
$\langle \mathbf{3}, \mathbf{1}, \mathbf{2} \rangle$
$\langle \mathbf{3}, \mathbf{2}, \mathbf{1} \rangle$

Capitolo 5

JSetL($\mathcal{FD}$)

In questo capitolo, dopo una breve introduzione alla libreria JSetL, verranno descritte le classi che implementano quanto visto nei capitoli precedenti.

Infine, ci sarà spazio per qualche esempio di programma che utilizza le nuove funzionalità di JSetL.

5.1 La libreria JSetL

JSetL[16][25] è una libreria Java, sviluppata presso il Dipartimento di Matematica dell'Università di Parma, che permette di combinare il paradigma *object-oriented* di Java con i concetti classici dei linguaggi CLP quali ad esempio variabili logiche, liste (anche parzialmente specificate), unificazione, risoluzione di vincoli, non-determinismo.

La libreria fornisce inoltre la possibilità di trattare insiemi e vincoli insiemistici seguendo l'approccio descritto in CLP($\mathcal{SET}$)[6].

L'unificazione può coinvolgere sia variabili logiche che liste e insiemi ('*set unification*'). I vincoli gestiti sono di vario genere: dai vincoli insiemistici di base (quali ad esempio appartenenza, unione, intersezione...) ad altri più generici come l'uguaglianza o la disuguaglianza.

La risoluzione dei vincoli in JSetL è inerentemente *non-deterministica*. In particolare, lo è la risoluzione dei vincoli insiemistici: le soluzioni vengono calcolate mediante l'utilizzo di *choice points* e *backtracking*.

Questa caratteristica permette di ottenere la completezza della risoluzione; tuttavia, il forte utilizzo del non-determinismo può causare tempi di calcolo non ragionevoli per la ricerca della soluzione.

Per cercare di superare questo scoglio, relativamente alla risoluzione di vincoli su interi, JSetL fornisce un risolutore[14] che segue un approccio molto

simile a quello visto nel precedente capitolo.
Tuttavia, vi sono alcune differenze fondamentali:

- il dominio delle variabili è un intervallo, anziché un multi-intervallo
- l'insieme delle regole di riduzione dei domini è più ristretto
- il labeling non fa uso di alcuna euristica: di fatto è un *Generate & Test*.

Questa implementazione verrà quindi arricchita e migliorata seguendo l'approccio definito nel capitolo precedente.

Mantenendo la stessa dicitura di [14], quest'estensione della libreria verrà denominata JSetL($\mathcal{FD}$).

Nelle prossime sezioni verrà quindi illustrato come JSetL($\mathcal{FD}$) permetta, attraverso appositi costrutti Java, di manipolare tutto quanto visto in precedenza: intervalli, multi-intervalli, simboli e vincoli del linguaggio $\mathcal{L}_{\mathcal{FD}}$, labeling.

5.2 La classe `Interval`

La classe `Interval` è una struttura dati che permette di rappresentare e manipolare gli intervalli di $\mathbb{I}_\alpha$.

Per prima cosa, conviene notare che gli estremi dell'universo $\mathbb{Z}_\alpha = [-\alpha..\alpha]$ sono definiti nel seguente modo:

- $\alpha =$ `Interval.`***`SUP`*** $\stackrel{def}{=}$ `Integer.`***`MAX_VALUE`*** `/ 2` $= 1073741823$
- $-\alpha =$ `Interval.`***`INF`*** $\stackrel{def}{=}$ `-Interval.`***`SUP`*** $= -1073741823$.

In questo modo, $|\mathbb{Z}_\alpha| = 2\alpha + 1 =$ `Integer.`***`MAX_VALUE`***: non si presentano problemi di *Integer overflow* nel calcolo di $|I|$, $I \oplus J$ e $I \ominus J$ per ogni $I, J \in \mathbb{I}_\alpha$. Potrebbero invece sorgere problemi nel calcolo di $I \odot J$ e $I \oslash J$: in questo caso si utilizzano opportuni *cast* a `Long` e `Double` rispettivamente.

Oltre ai campi statici ***`INF`*** e ***`SUP`***, la classe `Interval` fornisce il metodo statico ***`universe()`*** che ritorna l'universo $\mathbb{Z}_\alpha$.

Qui di seguito vengono illustrati i principali metodi della classe:

Costruttori

- `Interval()`: costruisce l'intervallo vuoto $\varnothing$
- `Interval(Integer a)`: costruisce l'intervallo $\|\{\texttt{a}\}\|_\alpha$
- `Interval(Integer a, Integer b)`: costruisce l'intervallo $[\![\texttt{a}..\texttt{b}]\!]_\alpha$.

Operatori insiemistici

- `boolean subset(Interval I)`: ritorna `true` sse `this` $\subseteq I$
- `Interval intersect(Interval I)`: restituisce l'intervallo `this` $\cap$ `I`
- `Interval sum(Interval I)`: restituisce l'intervallo `this` $\oplus$ `I`
- `Interval sub(Interval I)`: restituisce l'intervallo `this` $\ominus$ `I`
- `Interval opposite()`: restituisce l'intervallo $\ominus$`this` $\stackrel{def}{=}$ $\{0\} \ominus$ `this`.

Si noti che sono stati resi pubblici solamente i metodi corrispondenti ad operazioni che *non effettuano sovra-approssimazioni*; di conseguenza, non esistono metodi pubblici che implementano $\odot, \oslash, \cup_{\mathbb{I}}, \setminus_{\mathbb{I}}$ e $\sim_{\mathbb{I}}$.

Altri metodi di utilità

- `boolean contains(Integer k)`: ritorna `true` sse $k \in$ `this`
- `boolean isEmpty()`: ritorna `true` sse `this` $= \varnothing$
- `boolean isSingleton()`: ritorna `true` sse $|$`this`$| = 1$
- `boolean isUniverse()`: ritorna `true` sse `this` $= \mathbb{Z}_\alpha$
- `int size()`: ritorna $|$`this`$|$
- `Integer getGlb()`: ritorna `this`$^-$ se `this` $\neq \varnothing$, `null` altrimenti
- `Integer getLub()`: ritorna `this`$^+$ se `this` $\neq \varnothing$, `null` altrimenti
- `Iterator<Integer> iterator()`: ritorna un iteratore sugli elementi di `this`, in ordine crescente.

5.3 La classe `MultiInterval`

La classe `MultiInterval` permette di rappresentare e manipolare i multi-intervalli di $\mathbb{M}_\alpha$.

Come per la classe `Interval`, gli estremi dell'universo sono:

- $-\alpha =$ `MultiInterval.`*`INF`* $\stackrel{def}{=}$ `Interval.`*`INF`*
- $\alpha =$ `MultiInterval.`*`SUP`* $\stackrel{def}{=}$ `Interval.`*`SUP`*.

per cui `MultiInterval.`***`universe()`*** $=$ `Interval.`***`universe()`*** $= \mathbb{Z}_\alpha$.

Inoltre, questa classe implementa l'interfaccia Java `Set<Integer>`: per questo motivo deve essere fornita l'implementazione dei metodi (elencati in Figura 5.1) di `Set` e delle sue *super-interfacce* `Collection` e `Iterable`.

Qui di seguito vengono invece illustrati i principali metodi 'propri' della classe `MultiInterval`.

Costruttori

Oltre ai costruttori descritti in precedenza per la classe `Interval`, la classe `MultiInterval` offre i seguenti metodi:

- `MultiInterval(Set<Integer> S)` : costruisce il multi-intervallo corrispondente all'insieme $\|\mathtt{S}\|_\alpha$
- `MultiInterval(Collection<Interval> I)`: costruisce il multi-intervallo corrispondente a $I_1 \cup \ldots \cup I_n$, se `I` è una collezione di intervalli $[I_1, \ldots, I_n]$.

Operatori insiemistici

- `boolean subset(MultiInterval M)`: ritorna `true` sse `this` $\subseteq$ `M`
- `MultiInterval complement()`: ritorna $\sim_{\mathbb{M}}$(`this`)
- `MultiInterval union(MultiInterval M)`: ritorna `this` $\cup$ `M`
- `MultiInterval intersect(MultiInterval M)`: ritorna `this` $\cap$ `M`
- `MultiInterval diff(MultiInterval M)`: ritorna `this` $\setminus$ `M`
- `MultiInterval sum(MultiInterval M)`: ritorna `this` $\boxplus$ `M`
- `MultiInterval sub(MultiInterval M)`: ritorna `this` $\boxminus$ `M`
- `MultiInterval opposite()`: ritorna $\boxminus$`this` $\stackrel{def}{=}$ $\{0\} \boxminus$ `M`.

Si noti che, a causa della loro inefficienza, le operazioni di $\boxdot$ e $\boxslash$ non sono state implementate.

Method Summary	
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

Figura 5.1: Elenco dei metodi dell'interfaccia generica `java.util.Set<E>`

Altri metodi di utilità

Oltre ai metodi di utilità descritti in precedenza per la classe `Interval`, la classe `MultiInterval` offre i seguenti metodi:

- `Interval convexHull()`: ritorna $\mathcal{CH}_\alpha(\texttt{this})$
- `int getOrder()`: ritorna l'ordine di `this`, cioè il numero di intervalli disgiunti che lo compongono
- `boolean add(Integer k)`: modifica `this` in $\texttt{this}' = \|\texttt{this} \cup \{k\}\|_\alpha$. Ritorna `true` sse $\texttt{this}' \neq \texttt{this}$
- `boolean remove(Object x)`: modifica `this` in $\texttt{this}' = \|\texttt{this} \setminus \{x\}\|_\alpha$. Ritorna `true` sse $\texttt{this}' \neq \texttt{this}$.

5.4 La classe `FDVar`

La classe `FDVar` permette di modellare le variabili appartenenti a $\mathcal{V}_{\mathcal{FD}}$, oltre che altre espressioni ben-fondate dell'alfabeto del linguaggio $\mathcal{L}_{\mathcal{FD}}$ come termini e vincoli.

In JSetL, questa classe estende la *super-classe* `LVar` (utilizzata per rappresentare variabili logiche generiche) per poter rappresentare variabili aventi come dominio un insieme finito di interi contenuto in $\mathbb{Z}_\alpha$.
Vediamone i principali metodi.

Costruttori

Vengono fornite coppie di costruttori: ad ogni `FDVar` è infatti possibile associare un *nome esterno* opzionale.

- `FDVar()`: crea una nuova variabile, con dominio $\mathbb{Z}_\alpha$ e senza alcun nome esterno
- `FDVar(String s)`: come sopra, ma associa alla variabile il nome esterno `s`
- `FDVar(Integer a)`: crea una nuova variabile, con dominio $\|\{\texttt{a}\}\|_\alpha$ e senza alcun nome esterno
- `FDVar(String s, Integer a)`: come sopra, ma associa alla variabile il nome esterno `s`

- `FDVar(Integer a, Integer b)`: crea una nuova variabile, con dominio $[\![\texttt{a..b}]\!]_\alpha$ e senza alcun nome esterno
- `FDVar(String s, Integer a, Integer b)`: come sopra, ma associa alla variabile il nome esterno `s`
- `FDVar(MultiInterval M)`: crea una nuova variabile, con dominio `M` e senza alcun nome esterno
- `FDVar(String s, MultiInterval M)`: come sopra, ma associa alla variabile il nome esterno `s`.

Si noti che i costruttori potrebbero sollevare l'eccezione:

`NotValidDomainException`

nel caso si provi a costruire una variabile con dominio vuoto. In questo modo si anticipa un sicuro fallimento nella risoluzione di vincoli contenenti tali variabili.

5.4.1 Termini

La classe `FDVar` permette di costruire termini di $\mathcal{T}_{\mathcal{FD}}$, utilizzando le costanti intere della classe `Integer` e i seguenti metodi, corrispondenti ai simboli funzionali di $\mathcal{F}_{\mathcal{FD}}$:

- `FDVar sum(Integer k)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this} + \mathbf{k}$
- `FDVar sum(FDVar x)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this} + \mathbf{x}$
- `FDVar sub(Integer k)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this} - \mathbf{k}$
- `FDVar sub(FDVar x)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this} - \mathbf{x}$
- `FDVar mul(Integer k)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this} \cdot \mathbf{k}$
- `FDVar mul(FDVar x)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this} \cdot \mathbf{x}$
- `FDVar sum(Integer k)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this}/\mathbf{k}$
- `FDVar sum(FDVar x)`: ritorna una `FDVar y` tale che $\texttt{y} = \texttt{this}/\mathbf{x}$.

Come si può notare, non esiste un'apposita struttura dati per codificare i *termini composti* $\tau \in \mathcal{T}_{\mathcal{FD}} \setminus (\mathcal{V}_{\mathcal{FD}} \cup \mathcal{C}_{\mathcal{FD}})$: essi vengono restituiti sotto forma di *vincoli* del tipo $x = \tau$ con $x \in \mathcal{V}_{\mathcal{FD}}$.

Inoltre, si noti che l'ordine di precedenza degli operatori in un termine viene definito *implicitamente* dall'utente nel momento dell'invocazione dei metodi corrispondenti.
Ad esempio, il costrutto:

```
x.sum(y).mul(2);
```

permette di codificare il termine $(\mathtt{x} + \mathtt{y}) \cdot 2$, in quanto `x.sum(y)` ritorna una `FDVar v` tale che $\mathtt{v} = \mathtt{x} + \mathtt{y}$; quindi viene richiamata `v.mul(2)` che restituisce una `FDVar w` tale che $\mathtt{w} = \mathtt{v} \cdot 2 = (\mathtt{x} + \mathtt{y}) \cdot 2$.
Se invece volessi costruire il termine $\mathtt{x} + \mathtt{y} \cdot 2$ dovrei scrivere qualcosa come:

```
x.sum(y.mul(2));
```

che prima costruisce una `FDVar v` tale che $\mathtt{v} = \mathtt{y} \cdot 2$ quindi ritorna una `FDVar w` tale che $\mathtt{w} = \mathtt{x} + \mathtt{v} = \mathtt{x} + (\mathtt{y} \cdot 2)$.

5.4.2 Vincoli

I vincoli in JSetL sono codificati dalla classe `Constraint`.
Non ci soffermeremo troppo su questa classe; vale però la pena citare il metodo:

```
Constraint and(Constraint c)
```

che ritorna il vincolo composto `this` $\wedge$ `c`.
Vediamo invece come i seguenti metodi di `FDVar` permettano la costruzione di vincoli atomici appartenenti all'insieme $\mathcal{AC}_{\mathcal{FD}}$.

- `Constraint dom(Integer a, Integer b)`: ritorna il vincolo `this` :: [a..b]
- `Constraint dom(MultiInterval M)`: ritorna il vincolo `this` :: `M`
- `Constraint eq(Integer k)`: ritorna il vincolo `this` $=$ `k`
- `Constraint eq(FDVar x)`: ritorna il vincolo `this` $=$ `x`
- `Constraint neq(Integer k)`: ritorna il vincolo `this` $\neq$ `k`
- `Constraint neq(FDVar x)`: ritorna il vincolo `this` $\neq$ `x`
- `Constraint le(Integer k)`: ritorna il vincolo `this` $\leq$ `k`
- `Constraint le(FDVar x)`: ritorna il vincolo `this` $\leq$ `x`
- `Constraint lt(Integer k)`: ritorna il vincolo `this` $<$ `k`

- `Constraint lt(FDVar x)`: ritorna il vincolo `this` $<$ `x`
- `Constraint ge(Integer k)`: ritorna il vincolo `this` $\geq$ `k`
- `Constraint ge(FDVar x)`: ritorna il vincolo `this` $\geq$ `x`
- `Constraint gt(Integer k)`: ritorna il vincolo `this` $>$ `k`
- `Constraint gt(FDVar x)`: ritorna il vincolo `this` $>$ `x`.

Supponiamo ad esempio di voler costruire il vincolo `x` :: [`1..10`] $\wedge$ `x` $\neq$ `5`; il costrutto che restituisce il corrispondente `Constraint` è:

```
x.dom(1, 10).and(x.neq(5))
```

5.4.3 Labeling

Prima di descrivere i metodi riguardanti la gestione del labeling, è bene introdurre brevemente la classe `LabelingOptions`, che permette di impostare le euristiche di labeling settando opportunamente i campi `val` e `var` (viceversa, vengono mantenute le opzioni di *default*).
Se ad esempio volessi impostare ρ = first-fail e λ = median, dovrei scrivere qualcosa come:

```
LabelingOptions lop = new LabelingOptions();
lop.var = VarHeuristic.FIRST_FAIL;
lop.val = ValHeuristic.MEDIAN;
```

dove `ValHeuristic` e `VarHeuristic` sono *enumerazioni* Java che racchiudono le possibili euristiche ρ e λ:

```
public enum VarHeuristic {
    RIGHT_MOST,
    LEFT_MOST,
    MID_MOST,
    MIN,
    MAX,
    FIRST_FAIL,
    RANDOM,
}

public enum ValHeuristic {
    GLB,
    LUB,
```

```
    MID_MOST,
    MEDIAN,
    EQUI_RANDOM,
    RANGE_RANDOM,
    MID_RANDOM,
}
```

I valori di default sono `VarHeuristic = LEFT_MOST` e `ValHeuristic = GLB`.

Vediamo quindi i metodi che `FDVar` offre per il supporto del labeling:

- `Constraint label()`: forza il labeling su `this`, mantenendo le opzioni di default
- `Constraint label(ValHeuristic val)`: forza il labeling su `this`, utilizzando come euristica $\lambda =$ `val`
- `static Constraint` *`label`*`(AbstractList<FDVar> l)`: forza il labeling sulla lista di variabili `l`, mantenendo le opzioni di default
- `static Constraint` *`label`*`(FDVar[] l)`: come sopra, con `l` array
- `static Constraint` *`label`*`(AbstractList<FDVar> l, LabelingOptions lop)`: forza il labeling sulla lista di variabili `l`, utilizzando le opzioni di `lop`
- `static Constraint` *`label`*`(FDVar[] l, LabelingOptions lop)`: come sopra, con `l` array.

Si noti che tutti questi metodi ritornano un oggetto di tipo `Constraint`, in quanto le richieste di labeling su una o più variabili vengono a tutti gli effetti trattate come vincoli su di esse.

Altri metodi di utilità

Vediamo infine alcuni metodi ausiliari della classe `FDVar`.

- `Constraint ndom(Integer a, Integer b)`: ritorna il vincolo `this` $:: \sim_{\mathbb{M}}([a..b])$
- `Constraint ndom(MultiInterval M)`: ritorna il vincolo `this` $:: \sim_{\mathbb{M}}(M)$

- `static Constraint` ***allDifferent*** `(AbstractList<FDVar> l)`: ritorna il vincolo
$$\bigwedge_{1 \leq i < j \leq n} x_i \neq x_j$$
se `l` è una lista di `FDVar` $[x_1, \ldots, x_n]$

- `static Constraint` ***allDifferent*** `(FDVar[] l)`: come sopra, con `l` array

- `boolean equals(FDVar x)`: ritorna `true` sse `this` = `x`

- `MultiInterval getDomain()`: ritorna $D_{\texttt{this}}$

- `void output()`: stampa `this` su *standard output*, incluse informazioni sul suo dominio se $|D_{\texttt{this}}| > 1$.

5.5 Risoluzione dei vincoli

Occupiamoci finalmente del processo di risoluzione dei vincoli di $\mathcal{L}_{\mathcal{FD}}$.

In JSetL, il *constraint solver* è rappresentato dalla classe `SolverClass`. Ogni sua istanza contiene un *constraint store*, inizialmente vuoto, al quale è possibile aggiungere dinamicamente vincoli mediante il metodo:

```
void add(Constraint c)
```

Una volta costruiti i vincoli del problema da risolvere (utilizzando i corrispondenti metodi della classe `FDVar`) essi dovranno quindi essere inseriti nel constraint store del solver.

A questo punto è possibile avviare il processo di risoluzione descritto nel capitolo precedente, mediante l'invocazione del metodo:

```
void solve()
```

Tale metodo cercherà di ridurre i vincoli del constraint store in una forma logicamente equivalente, attraverso:

- riscritture dei vincoli in una delle forme descritte in 4.9
- applicazione delle regole dei riduzione dei domini viste in 4.3
- propagazione dei vincoli
- eventuale labeling sulle variabili.

Esistono apposite classi JSetL che si occupano di questo procedimento (ad esempio `RwRulesFD`, `DomainRulesFD`, `Backtracking` ecc..).
Tuttavia esse sono *trasparenti* all'utente, per cui non ci si occuperà di loro.

Si noti invece che il metodo `solve` non ritorna nulla: in caso di fallimento della risoluzione viene sollevata l'eccezione `Failure`.

In alternativa a `solve`, è possibile invocare il metodo:

```
boolean check()
```

che ritorna `false` se la risoluzione fallisce, `true` altrimenti (si noti ancora una volta che ciò non implica necessariamente la consistenza del CSP).

5.6 Esempi di programmi

Per concludere il capitolo, illustriamo qualche esempio completo di programmi che utilizzano le funzionalità descritte in precedenza.

5.6.1 Permutazioni

Riprendiamo l'esempio delle permutazioni descritto in 6, fornendo una sua possibile implementazione:

```
import java.util.Vector;
import JSetL.*;

class PermutationsFD {

    public static void main (String[] args)
    throws Failure {
        // Constraint solver.
        SolverClass solver = new SolverClass();

        // x, y, z::[1..3]
        FDVar x = new FDVar("x", 1, 3);
        FDVar y = new FDVar("y", 1, 3);
        FDVar z = new FDVar("z", 1, 3);

        // Labeling on [x, y].
        Vector<FDVar> v = new Vector<FDVar>(3);
        v.add(x);
        v.add(y);
```

```
        solver.add(FDVar.label(v));

        // x neq y AND y neq z AND z neq x.
        v.add(z);
        solver.add(FDVar.allDifferent(v));

        // Try to find a solution.
        solver.solve();

        // Print all solutions.
        int i = 0;
        do {
            ++i;
            System.out.println("Solution no. " + i);
            x.output();
            y.output();
            z.output();
        } while (solver.nextSolution());
    }
}
```

La semantica del programma è piuttosto intuitiva.

Vale la pena ricordare che, non avendo fornito alcuna opzione di labeling sulle variabili del vettore `v`, vengono mantenute le impostazioni di default (ρ = left-most e λ = glb) che sono proprio quelle dell'esempio 6.

Inoltre, il metodo

```
boolean nextSolution()
```

della classe `SolverClass` cerca di trovare una nuova soluzione; esso ritorna `true` se una tale soluzione esiste, `false` altrimenti.

L'output del programma `Permutation` sarà quindi:

```
Solution no. 1
x = 1
y = 2
z = 3
Solution no. 2
x = 1
y = 3
z = 2
Solution no. 3
```

```
x = 2
y = 1
z = 3
Solution no. 4
x = 2
y = 3
z = 1
Solution no. 5
x = 3
y = 1
z = 2
Solution no. 6
x = 3
y = 2
z = 1
```

5.6.2 SEND + MORE = MONEY

I problemi *cripto-aritmetici* sono problemi matematici costituiti da un insieme di equazioni nelle quali ad ogni lettera diversa corrisponde una certa cifra.
L'obiettivo è identificare il valore di ogni lettera in modo da soddisfare tali equazioni.

Il problema SEND + MORE = MONEY è un classico esempio di problema cripto-aritmetico: l'obiettivo è assegnare ad ogni lettera
$l \in \{S, E, N, D, M, O, R, Y\}$ una cifra $d_l \in [0..9]$ in modo che sia soddisfatta l'equazione:

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

Tale problema può quindi essere facilmente tradotto in un $CSP(\mathcal{FD})$ nel quale:

- $\mathcal{V} = \{S, E, N, D, M, O, R, Y\}$
- $D_x = [0..9]$ per ogni $x \in \mathcal{V}$
- $S \neq 0 \wedge M \neq 0$, in quanto tali variabili sono le cifre più significative di $SEND$, $MORE$ e $MONEY$
- tutte le variabili di $\mathcal{V}$ sono distinte

- deve valere la seguente equazione:

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

Vediamo quindi il programma che risolve questo CSP:

```
public static void main (String[] args)
throws IOException, Failure {
    FDVar s = new FDVar("S", 0, 9);
    FDVar e = new FDVar("E", 0, 9);
    FDVar n = new FDVar("N", 0, 9);
    FDVar d = new FDVar("D", 0, 9);
    FDVar m = new FDVar("M", 0, 9);
    FDVar o = new FDVar("O", 0, 9);
    FDVar r = new FDVar("R", 0, 9);
    FDVar y = new FDVar("Y", 0, 9);
    FDVar send = new FDVar("SEND");
    FDVar more = new FDVar("MORE");
    FDVar money = new FDVar("MONEY");
    FDVar[] letters = {s, e, n, d, m, o, r, y};
    SolverClass solver = new SolverClass();

    // S neq 0 AND M neq 0.
    solver.add(s.neq(0).and(m.neq(0)));

    // Each variable is different from each other.
    solver.add(FDVar.allDifferent(letters));

    // SEND = S*1000 + E*100 + N*10 + D.
    solver.add(send.eq(
        s.mul(1000).sum(e.mul(100).sum(n.mul(10).sum(d)))));
    // MORE = M*1000 + O*100 + R*10 + E.
    solver.add(more.eq(
        m.mul(1000).sum(o.mul(100).sum(r.mul(10).sum(e)))));
    // MONEY = M*10000 + O*1000 + N*100 + E*10 + Y.
        solver.add(money.eq(
        m.mul(10000).sum(o.mul(1000).sum(n.mul(100)
                    .sum(e.mul(10).sum(y))))));
    // MONEY = SEND + MORE.
```

```
    solver.add(money.eq(send.sum(more)));

    // Labeling on S and E variables.
    solver.add(s.label());
    solver.add(e.label());

    // Try to find a solution.
    solver.solve();
}
```

Si noti che per raggiungere la soluzione (unica) di questo problema non è sufficiente impostare i soli vincoli sopra elencati, ma è necessario forzare il labeling.

Tuttavia, non è necessario fare labeling su tutte le variabili di $\mathcal{V}$: è sufficiente farlo (senza l'utilizzo di particolari euristiche) su S ed E per trovare immediatamente la soluzione.

Nell'esempio è stata infine omessa la (poco significativa) parte relativa alla stampa della soluzione, che è:

```
 9567 <===>  SEND +
 1085 <===>  MORE =
------------------
10652 <===> MONEY
```

5.6.3 Sudoku

Come ultimo esempio, occupiamoci del popolare gioco del *Sudoku*.

In questo gioco di logica, al giocatore viene proposta una griglia di 9×9 celle, ciascuna delle quali può contenere un numero da 1 a 9, oppure essere vuota; la griglia è suddivisa in nove righe orizzontali, nove colonne verticali e nove sottogriglie, chiamate regioni, di 3×3 celle contigue. Le griglie proposte al giocatore hanno da 20 a 35 celle contenenti un numero. Scopo del gioco è quello di riempire le caselle bianche con numeri da 1 a 9, in modo tale che in ogni riga, colonna e regione siano presenti tutte le cifre da 1 a 9 senza ripetizioni.

Questo gioco può quindi essere formalizzato da un $CSP(\mathcal{FD})$ nel quale:

- $\mathcal{V} = \{x_{ij} : 0 \leq i, j \leq 9\}$
- $D_x = [1..9]$ per ogni $x \in \mathcal{V}$
- se $x, y \in \mathcal{V}$ appartengono alla stessa riga, alla stessa colonna o alla stessa regione allora $x \neq y$.

Un matematico finlandese, Arto Inkala, ha realizzato il 'Sudoku più difficile al mondo' (Figura 5.2).

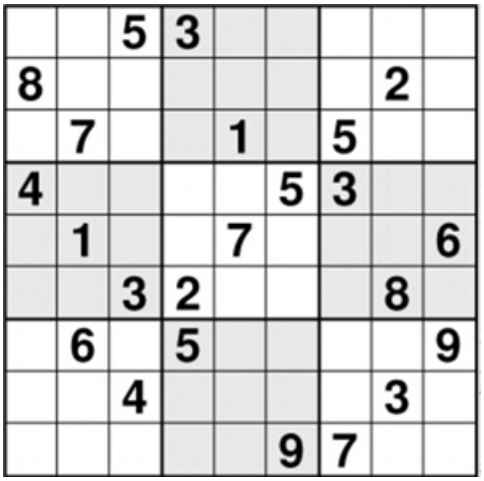


Figura 5.2: il Sudoku più difficile al mondo, secondo Arto Inkala.

Mostriamo quindi come il seguente programma sia in grado di risolverlo istantaneamente, utilizzando le funzionalità di JSetL($\mathcal{FD}$).

```
public static void main(String[] args) {
    FDVar[][] board = new FDVar[9][9];
    SolverClass solver = new SolverClass();
    int i, j;

    for (i = 0; i < 9; ++i) {
        FDVar[] row = new FDVar[9];
        for (j = 0; j < 9; ++j) {
            FDVar tmp = new FDVar(1, 9);
            board[i][j] = tmp;
            row[j] = board[i][j];
        }
        // All the numbers in the same row must be different.
        solver.add(FDVar.allDifferent(row));
    }

    for (j = 0; j < 9; ++j) {
        FDVar[] column = new FDVar[9];
        for (i = 0; i < 9; ++i)
```

```
        column[i] = board[i][j];
        // All the numbers in the same column must be different.
        solver.add(FDVar.allDifferent(column));
    }

    i = 0;
    j = 0;
    for (int h = 0; h < 9; ++h) {
        FDVar[] region = new FDVar[9];
        for (int k = 0; k < 9; ++k) {
            region[k] = board[i][j];
            ++j;
            if (j % 3 == 0) {
                ++i;
                j -= 3;
            }
        }
        if (i == 9) {
            i = 0;
            j += 3;
        }
        // All the numbers in the same region must be different.
        solver.add(FDVar.allDifferent(region));
    }

    // labeling on {board[i][j]: 1 <= i <= 3, 0 <= j <= 3}
    for (i = 1; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            solver.add(board[i][j].label());

    // The world's hardest sudoku puzzle, according to Arto Inkala.
    solver.add(board[0][2].eq(5));
    solver.add(board[0][3].eq(3));
    solver.add(board[1][0].eq(8));
    solver.add(board[1][7].eq(2));
    solver.add(board[2][1].eq(7));
    solver.add(board[2][4].eq(1));
    solver.add(board[2][6].eq(5));
    solver.add(board[3][0].eq(4));
    solver.add(board[3][5].eq(5));
    solver.add(board[3][6].eq(3));
```

```
    solver.add(board[4][1].eq(1));
    solver.add(board[4][4].eq(7));
    solver.add(board[4][8].eq(6));
    solver.add(board[5][2].eq(3));
    solver.add(board[5][3].eq(2));
    solver.add(board[5][7].eq(8));
    solver.add(board[6][1].eq(6));
    solver.add(board[6][3].eq(5));
    solver.add(board[6][8].eq(9));
    solver.add(board[7][2].eq(4));
    solver.add(board[7][7].eq(3));
    solver.add(board[8][5].eq(9));
    solver.add(board[8][6].eq(7));

    // Try to find a solution.
    solver.check();

    // Prints the solution.
    System.out.println(" =======================");
    for (i = 0; i < 9; ++i) {
        System.out.print("| ");
        for (j = 0; j < 9; ++j) {
            System.out.print(board[i][j] + " ");
            if (j % 3 == 2)
                System.out.print("| ");
        }
        System.out.println();
        if (i % 3 == 2)
            System.out.println(" =======================");
    }
}
```

Si noti ancora una volta come non sia necessario imporre il labeling su tutte le $9 \cdot 9 = 81$ variabili del problema.
Nel nostro caso, è sufficiente farlo ad esempio sulle variabili $x_{ij} \in \mathcal{V}$ tali che $1 \leq i \leq 3$ e $0 \leq j \leq 3$.

Ecco infine la soluzione, così come viene stampata dal programma:

```
 =======================
| 1 4 5 | 3 2 7 | 6 9 8 |
| 8 3 9 | 6 5 4 | 1 2 7 |
| 6 7 2 | 9 1 8 | 5 4 3 |
 =======================
| 4 9 6 | 1 8 5 | 3 7 2 |
| 2 1 8 | 4 7 3 | 9 5 6 |
| 7 5 3 | 2 9 6 | 4 8 1 |
 =======================
| 3 6 7 | 5 4 2 | 8 1 9 |
| 9 8 4 | 7 6 1 | 2 3 5 |
| 5 2 1 | 8 3 9 | 7 6 4 |
 =======================
```

Capitolo 6

Intervalli di insiemi di interi

In questo capitolo verrà discusso in modo formale un particolare dominio, analogamente a quanto fatto nel capitolo 3, che verrà poi utilizzato per modellare il dominio di variabili contenute in vincoli su insiemi di interi.

6.1 Set-interval

Definizione 6.1 (Set-interval)**.** Sia $\mathcal{P}(\mathbb{Z})$ l'insieme delle parti di $\mathbb{Z}$ e siano $A, B \subseteq \mathbb{Z}$. L'*intervallo di insiemi di interi* o, più brevemente, il **set-interval** di estremi A e B è il *sottoinsieme finito* $[A..B] \subseteq \mathcal{P}(\mathbb{Z})$ tale che:

$$[A..B] \stackrel{def}{=} \{X \subseteq \mathbb{Z} : A \subseteq X \subseteq B\}$$

A è il GLB, mentre B è il LUB dell'intervallo.

Osservazione 9. Siano $A, B \subseteq \mathbb{Z}$. Allora,

- se $A \nsubseteq B$, si ha che $[A..B] = \varnothing$
- se $A = B$, si ha che $[A..B] = [A..A] = \{A\}$
- se $A \subseteq B$, si ha che $[A..B]$ ha esattamente $2^{|B|-|A|}$ elementi.

Definizione 6.2 (Insieme $\mathbb{S}_\beta$)**.** Sia $\beta \geq 0$ una *fissata costante intera*[1] e sia $\mathbb{Z}_\beta \stackrel{def}{=} [-\beta..\beta]$. D'ora in avanti indicheremo con $\mathbb{S}_\beta$ l'insieme di tutti i set-interval i cui estremi appartengono all'*universo* $\mathcal{P}(\mathbb{Z}_\beta) = [\varnothing..\mathbb{Z}_\beta]$, cioè:

$$\mathbb{S}_\beta \stackrel{def}{=} \{[A..B] : A, B \subseteq \mathbb{Z}_\beta\}$$

[1] Come per la costante α introdotta nel capitolo 3, β assume la semantica della costante ∞, per cui si considera il suo valore come '*grande a piacere*'. Si noti che, per non ledere la generalità, sono stati scelti due nomi differenti per tali costanti; tuttavia, nulla vieta che sia $\alpha = \beta$.

Inoltre, preso un generico $S \in \mathbb{S}_\beta$, verrà indicato con

- S^- il greatest lower bound di S
- S^+ il least upper bound di S.

Osservazione 10 (Reticolo di Boole). Ogni set-interval non vuoto $S \in \mathbb{S}_\beta$ definisce un **reticolo di Boole** $\mathfrak{B}$ definito come:

$$\mathfrak{B} \stackrel{def}{=} (S; \subseteq; \cup; \cap; \approx; S^-; S^+)$$

dove $\subseteq, \cup, \cap$ sono le consuete operazioni insiemistiche su $\mathcal{P}(\mathbb{Z})$ e $\approx$ è tale che $\approx X \stackrel{def}{=} (S^+ \setminus X) \cup S^-$ per ogni $X \in S$.

Dimostrazione. Va provato che:

(i) $\mathfrak{B}' = (S; \subseteq; \cup; \cap)$ è un *reticolo distributivo*

(ii) $S^- = \min_\subseteq(S)$ e $S^+ = \max_\subseteq(S)$

(iii) $X \cup \approx X = S^+$ e $X \cap \approx X = S^-$.

Procediamo quindi con ordine:

(i) $\mathfrak{B}'$ è reticolo distributivo, in quanto:

- $\subseteq$ è una relazione di *ordine parziale* (riflessiva, antisimmetrica e transitiva) sull'insieme S
- $X \cup Y = Sup_\subseteq\{X, Y\}$ e $X \cap Y = Inf_\subseteq\{X, Y\}$ per ogni $X, Y \in S$
- $X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z)$ e $X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z)$ per ogni $X, Y, Z \in S$.

(ii) $S^- = min_\subseteq(S)$ e $S^+ = max_\subseteq(S)$ per definizione di set-interval

(iii) $X \cup \approx X = X \cup ((S^+ \setminus X) \cup S^-) = (X \cup (S^+ \setminus X)) \cup S^- = S^+ \cup S^- = S^+$
$X \cap \approx X = X \cap ((S^+ \setminus X) \cup S^-) = (X \cap (S^+ \setminus X)) \cup (X \cap S^-) = \varnothing \cup S^- = S^-$.

■

Un set-interval $S \in \mathbb{S}_\beta$ può quindi essere rappresentato da un *diagramma di Hasse*, un grafo orientato nel quale:

- esiste un unico nodo N_X per ogni sottoinsieme $X \in S$
- esiste un arco $N_X \to N_Y$ se e solo se $X = max_\subseteq\{Z \in S : Z \subset Y\}$.

Sia ad esempio $S = [A..B]$ con $A = \{0\}$ e $B = \{-3, -1, 0, 2\}$.
S ha esattamente $2^{4-1} = 8$ elementi, rappresentati dal diagramma di Hasse illustrato in figura 6.1.

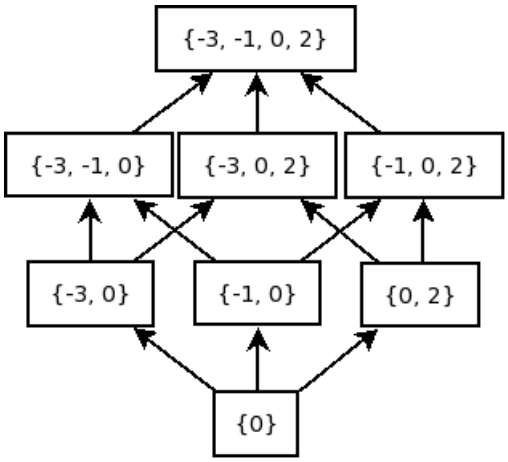


Figura 6.1: Diagramma di Hasse del set-interval $[\{0\}..\{-3,-1,0,2\}]$

6.2 Operazioni sui set-interval

Definizione 6.3 (Normalizzazione)**.** Sia $D \subseteq \mathcal{P}(\mathbb{Z})$. Definiamo operatore di **normalizzazione** (*rispetto a* $\mathcal{P}(\mathbb{Z}_\beta)$) l'applicazione $\|\cdot\|_\beta : \mathcal{P}^2(\mathbb{Z}) \longrightarrow \mathcal{P}^2(\mathbb{Z}_\beta)$ tale che:

$$\|D\|_\beta \stackrel{def}{=} D \cap \mathcal{P}(\mathbb{Z}_\beta)$$

D'ora in avanti, la normalizzazione di un set-interval $[A..B]$ verrà indicata con $[\![A..B]\!]_\beta$ anziché $\|[A..B]\|_\beta$.

Definizione 6.4 (Chiusura convessa)**.** Sia $D \subseteq \mathcal{P}(\mathbb{Z})$. Definiamo operatore di **chiusura convessa** (*rispetto a* $\mathcal{P}(\mathbb{Z}_\beta)$) l'applicazione $\mathcal{CH}_\beta : \mathcal{P}^2(\mathbb{Z}) \longrightarrow \mathbb{S}_\beta$ tale che:

$$\mathcal{CH}_\beta(D) \stackrel{def}{=} min_\subseteq\{S \in \mathbb{S}_\beta : \|D\|_\beta \subseteq S\}$$

Prendiamo ad esempio l'insieme $D = \{\{0\}, \{1\}, [2..\beta+1]\}$.
Essendo $[2..\beta+1] \notin \mathcal{P}(\mathbb{Z}_\beta)$, abbiamo che:

$$\|D\|_\beta = D \cap \mathcal{P}(\mathbb{Z}_\beta) = \{\{0\}, \{1\}\}$$

mentre

$$\mathcal{CH}_\beta(D) \stackrel{def}{=} min_\subseteq\{S \in \mathbb{S}_\beta : \{\{0\}, \{1\}\} \subseteq S\} = [\varnothing..\{0,1\}]$$

Si può quindi intuire che $\mathbb{S}_\beta$ *non è isomorfo a* $\mathcal{P}^2(\mathbb{Z}_\beta)$, come dimostrato nella seguente proposizione.

Proposizione 6.2.1. $\mathbb{S}_\beta \subset \mathcal{P}^2(\mathbb{Z}_\beta)$

Dimostrazione. Per definizione, $\mathbb{S}_\beta \subseteq \mathcal{P}^2(\mathbb{Z}_\beta)$. Esistono però elementi di $\mathcal{P}^2(\mathbb{Z}_\beta)$ che non appartengono a $\mathbb{S}_\beta$, basti pensare ad esempio all'insieme di insiemi $\{\{0\}, \{1\}\}$.

■

Osserviamo dunque come i set-interval presentino diverse analogie con gli intervalli di interi introdotti nel capitolo 3.

Anzitutto, sono *insiemi finiti* appartenenti ad un fissato *universo finito*. Per questo motivo, essi necessitano di un operatore di normalizzazione $\|\cdot\|_\beta$ per evitare di 'uscire' da tale universo.

Inoltre, come gli intervalli, i set-interval sono *insiemi convessi* di elementi: preso un generico $S \in \mathbb{S}_\beta$, per ogni $X, Y \in S$ si ha che:

$$\{Z : (X \cap Y) \subseteq Z \subseteq (X \cup Y)\} \subseteq S$$

Per questa ragione è necessario utilizzare un operatore di chiusura convessa $\mathcal{CH}_\beta$ per poter sovra-approssimare insiemi qualsiasi di $\mathcal{P}^2(\mathbb{Z}_\beta)$ (così come per gli intervalli era necessario usare $\mathcal{CH}_\alpha$ per approssimare insiemi qualsiasi di $\mathcal{P}(\mathbb{Z}_\alpha)$).

Naturalmente, queste approssimazioni comportano una perdita di precisione.
Si prenda ad esempio $D = \{\{x\} : x \in \mathbb{Z}_\beta\}$. In questo caso, $\|D\|_\beta = D$. Tuttavia, $D \notin \mathbb{S}_\beta$ quindi va considerata la sua chiusura convessa:

$$\mathcal{CH}_\beta(D) = min_\subseteq\{S \in \mathbb{S}_\beta : D \subseteq S\} = [\varnothing..\mathbb{Z}_\beta]$$

L'insieme risultante $[\varnothing..\mathbb{Z}_\beta]$ contiene $2^{2\beta+1}$ elementi anziché $2\beta + 1$: la differenza è notevole.

Analogamente agli intervalli, le uniche operazioni insiemistiche significative sui set-interval sono l'**intersezione**, per la quale per ogni $S, T \in \mathbb{S}_\beta$ si ha che:

$$S \cap T = [S^- \cup T^-..S^+ \cap T^+] \in \mathbb{S}_\beta$$

e l'**inclusione**, per la quale si ha che:

$$S \subseteq T \iff T^- \subseteq S^- \text{ e } S^+ \subseteq T^+$$

mentre per quanto riguarda unione, differenza insiemistica e complementazione l'utilizzo di normalizzazione e chiusura convessa comporta una perdita di informazione e proprietà elementari.

In particolare, dati $S \in \mathbb{S}_\beta$ e $X \in S$, si ha che $T = S \setminus \{X\}$ non è in generale un set-interval (intuitivamente quest'operazione si può interpretare come la rimozione del nodo N_X dal diagramma di Hasse di S).

La stessa cosa possiamo dire per gli intervalli di interi: presi $I \in \mathbb{I}_\alpha$ e $x \in I$, si ha che $I \setminus \{x\}$ non è in generale un intervallo. Tuttavia, l'operazione $I \setminus \{x\}$ individua esattamente due intervalli (eventualmente vuoti) $J = [I^-..x-1]$ e $K = [x+1..I^+]$ tali che $I \setminus \{x\} = J \cup K$.
Un'analoga operazione non è possibile per i set-interval: essi si basano sull'insieme *parzialmente ordinato* $(\mathcal{P}(\mathbb{Z}), \subseteq)$ mentre gli intervalli sull'insieme *totalmente ordinato* $(\mathbb{Z}, \leq)$.

Per questo motivo, la definizione di un dominio '*multi-set-interval*' analogo ai multi-intervalli, seppur possibile non sarebbe per nulla banale.

Anzitutto, su $\mathbb{S}_\beta$ non sarebbe possibile definire una relazione d'ordine stretto $\prec$ corrispondente a quella introdotta in 3.2: ci si deve accontentare di definire un multi-set-interval come l'unione di set-interval tra loro *disgiunti*.

Inoltre, essendo $\subseteq$ parziale, un insieme $D \subseteq \mathcal{P}(\mathbb{Z}_\beta)$ può essere partizionato nell'unione di set-interval disgiunti in modo *non univoco*. Si consideri ad esempio $D = \{\{1\}, \{2\}, \{1,2\}\}$. Tale insieme può essere partizionato in:

- $P_1 = \{\{1\}\} \cup \{\{2\}\} \cup \{\{1,2\}\}$
- $P_2 = \{\{1\}\} \cup [\{2\}..\{1,2\}]$
- $P_3 = \{\{2\}\} \cup [\{1\}..\{1,2\}]$

Un possibile approccio potrebbe essere quello di considerare fra le possibili partizioni quelle con cardinalità minore (nel nostro esempio, P_2 e P_3) e a questo punto definire un certo *ordinamento totale* $\preceq$ fra esse, in modo da scegliere poi la partizione minima secondo $\preceq$.

E' evidente che questi passaggi sono molto più costosi rispetto al partizionamento (osservato nella dimostrazione del teorema 3.2) di un insieme di interi nel corrispondente multi-intervallo.

Capitolo 7

Risoluzione di vincoli su insiemi di interi

In questo capitolo verranno introdotti due linguaggi e la relativa semantica: $\mathcal{L}_{\mathcal{FS}}$ e la sua estensione $\mathcal{L}_{\mathcal{FDS}}$. Quindi, verranno descritti regole e algoritmi per risolvere CSP basati su $\mathcal{L}_{\mathcal{FDS}}$, seguendo un approccio simmetrico rispetto a quanto fatto nel capitolo 4.

7.1 Il linguaggio $\mathcal{L}_{\mathcal{FS}}$

Definizione 7.1 (Alfabeto di $\mathcal{L}_{\mathcal{FS}}$)**.** L'**alfabeto** del linguaggio $\mathcal{L}_{\mathcal{FS}}$ è costituito da:

- un insieme *infinito* di **variabili** $\mathcal{V}_{\mathcal{FS}} \stackrel{def}{=} \{X, Y, Z, \ldots\}$
- il simbolo di **congiunzione logica** $\wedge$
- le **parentesi** tonde (e)
- il predicato binario di **dominio** ::
- un insieme di **predicati binari** $\mathcal{P}_{\mathcal{FS}} \stackrel{def}{=} \{=, \neq, \subset, \subseteq, ||\}$
- un insieme di **simboli funzionali** $\mathcal{F}_{\mathcal{FS}} \stackrel{def}{=} \{\sim, \cup, \cap, \setminus\}$
- un insieme *più che numerabile* di **costanti insiemistiche** $\mathcal{Z}_{\mathcal{FS}} \stackrel{def}{=} \{A : A \subseteq \mathbb{Z}\}$.

Definizione 7.2 (Termini di $\mathcal{L}_{\mathcal{FS}}$)**.** L'insieme $\mathcal{T}_{\mathcal{FS}}$ dei **termini** del linguaggio $\mathcal{L}_{\mathcal{FS}}$ è così definito:

(i) ogni variabile è un termine: $\mathcal{V}_{\mathcal{FS}} \subseteq \mathcal{T}_{\mathcal{FS}}$

(ii) ogni costante è un termine: $\mathcal{Z}_{\mathcal{FS}} \subseteq \mathcal{T}_{\mathcal{FS}}$

(iii) se $t \in \mathcal{T}_{\mathcal{FS}}$ allora $\sim t \in \mathcal{T}_{\mathcal{FS}}$

(iv) se $t_1, t_2 \in \mathcal{T}_{\mathcal{FS}}$ e $* \in \{\cup, \cap, \setminus\}$, allora $(t_1 * t_2) \in \mathcal{T}_{\mathcal{FS}}$

(v) nient'altro appartiene a $\mathcal{T}_{\mathcal{FS}}$.

Definizione 7.3 (Vincoli atomici di $\mathcal{L}_{\mathcal{FS}}$)**.** L'insieme $\mathcal{AC}_{\mathcal{FS}}$ dei **vincoli atomici** del linguaggio $\mathcal{L}_{\mathcal{FS}}$ è così definito:

(i) se $t \in \mathcal{T}_{\mathcal{FS}}$ e $S \in \mathbb{S}_\beta$ allora $t :: S \in \mathcal{AC}_{\mathcal{FS}}$

(ii) se $t_1, t_2 \in \mathcal{T}_{\mathcal{FS}}$ e $* \in \mathcal{P}_{\mathcal{FS}}$ allora $t_1 * t_2 \in \mathcal{AC}_{\mathcal{FS}}$

(iii) nient'altro appartiene a $\mathcal{AC}_{\mathcal{FS}}$.

Definizione 7.4 (Vincoli di $\mathcal{L}_{\mathcal{FS}}$)**.** L'insieme $\mathcal{C}_{\mathcal{FS}}$ dei **vincoli** del linguaggio $\mathcal{L}_{\mathcal{FS}}$ è così definito:

(i) ogni vincolo atomico è un vincolo: $\mathcal{AC}_{\mathcal{FS}} \subseteq \mathcal{C}_{\mathcal{FS}}$

(ii) se $c_1, c_2 \in \mathcal{C}_{\mathcal{FS}}$ allora $c_1 \wedge c_2 \in \mathcal{C}_{\mathcal{FS}}$

(iii) nient'altro appartiene a $\mathcal{C}_{\mathcal{FS}}$.

Esempio 7 (Termini e vincoli $\mathcal{L}_{\mathcal{FS}}$). Si nota facilmente come la definizione di $\mathcal{L}_{\mathcal{FS}}$ sia del tutto analoga a quella di $\mathcal{L}_{\mathcal{FD}}$, fatta eccezione per i simboli '*extra-logici*': predicati, funzioni e costanti.

Come per $\mathcal{L}_{\mathcal{FD}}$, riportiamo una piccola casistica di elementi che (non) appartengono agli insiemi sopracitati.
Sono ad esempio termini le seguenti espressioni:[1]

- $X \cap \{1, 2, 3\}$
- $(\mathbb{Z} \setminus Y) \cup \varnothing$
- $\sim Z \cap (X \cup Y)$

mentre non lo sono:

- $X \neq Y \cup \{1\}$

[1]Come per $\mathcal{L}_{\mathcal{FD}}$, d'ora in avanti verranno omesse le parentesi 'non significative': per evitare ambiguità si utilizzerà l'ordine di precedenza 'classico' degli operatori di $\mathcal{F}_{\mathcal{FS}}$, associando a sinistra.

- $X \setminus X = \varnothing$.

Sono invece vincoli le seguenti espressioni:

- $X :: [\{1\}..[1..4]]$
- $(X \cup Y) \cap Z \subseteq (X \cap Z) \cup (Y \cap Z)$
- $X = \{1, 2\} \wedge Y \subset \{0, 5\} \wedge X \,||\, Y$

7.1.1 Semantica del linguaggio $\mathcal{L}_{\mathcal{FS}}$

Per conferire significato ai simboli di $\mathcal{L}_{\mathcal{FS}}$, come in 4.1.1 utilizzeremo una *interpretazione* 'naturale' Σ su $\mathcal{P}(\mathbb{Z})$ che associa:

- ai predicati $P \in \mathcal{P}_{\mathcal{FS}}$, la relazione $P^\Sigma \subseteq \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z})$ tale che $P^\Sigma \stackrel{def}{=} P \restriction_{\mathcal{P}(\mathbb{Z})\times\mathcal{P}(\mathbb{Z})}$.[2]
- al simbolo funzionale di *complementazione insiemistica* $\sim$ l'operazione $\sim^\Sigma$: $\mathcal{P}(\mathbb{Z}) \longrightarrow \mathcal{P}(\mathbb{Z})$ tale che per ogni $X \in \mathcal{P}(\mathbb{Z})$

$$\sim^\Sigma(X) \stackrel{def}{=} \mathbb{Z}_\beta \setminus X$$

- ai simboli funzionali $f \in \mathcal{F}_{\mathcal{FS}} \setminus \{\sim\}$, l'operazione insiemistica $f^\Sigma : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \longrightarrow \mathcal{P}(\mathbb{Z})$ tale che $f^\Sigma \stackrel{def}{=} f \restriction_{\mathcal{P}(\mathbb{Z})\times\mathcal{P}(\mathbb{Z})}$
- alle costanti $A \in \mathcal{Z}_{\mathcal{FS}}$, l'insieme $A^\Sigma \stackrel{def}{=} A \in \mathcal{P}(\mathbb{Z})$.

Fissata questa interpretazione, possiamo definire formalmente il concetto di *soddisfacimento di vincoli* in modo analogo a quanto fatto nel capitolo 4.

Definizione 7.5 (Funzione di assegnazione)**.** Sia $\mathbb{Z}_\beta$ l'insieme universo definito in 6.2. Si definisce (*funzione di*) **assegnazione** su $\mathcal{V}_{\mathcal{FS}}$ una qualsiasi funzione $\varphi : \mathcal{V}_{\mathcal{FS}} \longrightarrow \mathcal{P}(\mathbb{Z}_\beta)$. Una volta definita tale φ, è possibile ricavarne la sua **estensione ai termini** $\varphi^\Sigma : \mathcal{T}_{\mathcal{FS}} \longrightarrow \mathcal{P}(\mathbb{Z})$ tale che:

$$\begin{array}{ll}
\varphi^\Sigma(X) = \varphi(X), & \text{per ogni } X \in \mathcal{V}_{\mathcal{FS}} \\
\varphi^\Sigma(A) = A^\Sigma, & \text{per ogni } A \in \mathcal{Z}_{\mathcal{FS}} \\
\varphi^\Sigma(\sim t) = \sim^\Sigma(\varphi^\Sigma(t)), & \text{per ogni } t \in \mathcal{T}_{\mathcal{FS}} \\
\varphi^\Sigma(t_1 * t_2) = \varphi^\Sigma(t_1) *^\Sigma \varphi^\Sigma(t_2), & \text{per ogni } t_1, t_2 \in \mathcal{T}_{\mathcal{FS}} \text{ e} \\
 & \text{per ogni } * \in \mathcal{F}_{\mathcal{FS}} \setminus \{\sim\}
\end{array}$$

[2] In particolare, il predicato di *disgiunzione insiemistica* $||$ sarà interpretato nella relazione $||^\Sigma \subseteq \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z})$ tale che $X, Y \in ||^\Sigma \iff X \cap Y = \varnothing$ per ogni $X, Y \in \mathcal{P}(\mathbb{Z})$

Definizione 7.6 (Soddisfacimento di vincoli)**.** Sia $c \in \mathcal{C}_{\mathcal{FS}}$ e φ una funzione di assegnazione su $\mathcal{V}_{\mathcal{FS}}$. Allora,

$$\begin{array}{lcll} \models_\varphi t :: S & \stackrel{def}{\Longleftrightarrow} & \varphi^\Sigma(t) \in S, & \text{per ogni } t \in \mathcal{T}_{\mathcal{FS}}, S \in \mathbb{S}_\beta \\ \models_\varphi t_1 * t_2 & \stackrel{def}{\Longleftrightarrow} & \varphi^\Sigma(t_1) *^\Sigma \varphi^\Sigma(t_2), & \text{per ogni } t_1, t_2 \in \mathcal{T}_{\mathcal{FS}}, * \in \mathcal{P}_{\mathcal{FS}} \\ \models_\varphi c_1 \wedge \ldots \wedge c_k & \stackrel{def}{\Longleftrightarrow} & \underset{i=1,\ldots,k}{\forall} \models_\varphi c_i, & \text{per ogni } c_1, \ldots, c_k \in \mathcal{C}_{\mathcal{FS}} \end{array}$$

Diremo quindi che un vincolo $c \in \mathcal{C}_{\mathcal{FS}}$ (non) è **soddisfacibile** se (non) esiste un'assegnazione $\varphi : \mathcal{V}_{\mathcal{FS}} \longrightarrow \mathcal{P}(\mathbb{Z}_\beta)$ che lo soddisfa, cioè tale che $\models_\varphi c$.

Esempio 8 (Soddisfacimento di vincoli). Consideriamo il vincolo:

$$X \subseteq [1..10] \wedge X :: [\{1,5\}..[1..50]]$$

Tale vincolo è soddisfacibile, basta prendere un'assegnazione φ tale che $\{1,5\} \subseteq \varphi(X) \subseteq [1..10]$. Ad esempio, presa φ tale che $\varphi(X) = \{1,5,8\}$ si ha che:

$$\begin{array}{rl} \models_\varphi X \subseteq [1..10] \wedge X :: [\{1,5\}..[1..50]] & \Longleftrightarrow \\ \models_\varphi X \subseteq [1..10] \text{ e } \models_\varphi X :: [\{1,5\}..[1..50]] & \Longleftrightarrow \\ \varphi^\Sigma(X) \subseteq [1..10] \text{ e } \varphi^\Sigma(X) \in [\{1,5\}..[1..50]] & \Longleftrightarrow \\ \varphi(X) \subseteq [1..10] \text{ e } \varphi(X) \in [\{1,5\}..[1..50]] & \Longleftrightarrow \\ \{1,5,8\} \subseteq [1..10] \text{ e } \{1,5,8\} \in [\{1,5\}..[1..50]] & \end{array}$$

Viceversa, si prenda il vincolo $X = \{0, \beta+1\}$. Nell'interpretazione che abbiamo fornito, tale vincolo non è soddisfacibile. Infatti, non esiste alcuna $\varphi : \mathcal{V}_{\mathcal{FS}} \longrightarrow \mathcal{P}(\mathbb{Z}_\beta)$ tale che $\varphi^\Sigma(X) = \{0, \beta+1\}$, in quanto per definizione $\varphi^\Sigma(X) = \varphi(X) \in \mathcal{P}(\mathbb{Z}_\beta)$ ma $\{0, \beta+1\} \not\subseteq \mathbb{Z}_\beta$ perciò $\{0, \beta+1\} \notin \mathcal{P}(\mathbb{Z}_\beta)$.

Per concludere, le nozioni (e le rispettive notazioni) di **conseguenza** ed **equivalenza logica** di (insiemi di) vincoli appartenenti a $\mathcal{C}_{\mathcal{FS}}$ sono esattamente le stesse date in 4.7 per i vincoli di $\mathcal{C}_{\mathcal{FD}}$.

7.2 Il linguaggio $\mathcal{L}_{\mathcal{FDS}}$

Sebbene $\mathcal{L}_{\mathcal{FS}}$ sia sufficiente per definire un insieme di regole di inferenza per la risoluzione di CSP basati su tale linguaggio, in questa sezione verrà fornita una sua *estensione* che lo arricchirà notevolmente.

Esso sarà infatti ampliato mediante l'aggiunta dei simboli di $\mathcal{L}_{\mathcal{FD}}$ e di nuovi simboli extra-logici come $\#, \in, \notin$.

Questo nuovo linguaggio, denominato $\mathcal{L}_{\mathcal{FDS}}$, permetterà di aumentare il potere espressivo e migliorare notevolmente la risoluzione di CSP contenenti vincoli su interi ed insiemi di interi.

Vediamo quindi com'è definito formalmente tale linguaggio.

Definizione 7.7 (Alfabeto di $\mathcal{L}_{\mathcal{FDS}}$)**.** L'**alfabeto** del linguaggio $\mathcal{L}_{\mathcal{FDS}}$ è costituito da:

- un insieme *infinito* di **variabili** $\mathcal{V}_{\mathcal{FDS}} \stackrel{def}{=} \mathcal{V}_{\mathcal{FD}} \cup \mathcal{V}_{\mathcal{FS}}$
- il simbolo di **congiunzione logica** $\wedge$
- le **parentesi** tonde (e)
- il predicato binario di **dominio** ::
- un insieme di **predicati** $\mathcal{P}_{\mathcal{FDS}} \stackrel{def}{=} \mathcal{P}_{\mathcal{FD}} \cup \mathcal{P}_{\mathcal{FS}} \cup \{\in, \notin\}$
- un insieme di **simboli funzionali** $\mathcal{F}_{\mathcal{FDS}} \stackrel{def}{=} \mathcal{F}_{\mathcal{FD}} \cup \mathcal{F}_{\mathcal{FS}} \cup \{\#\}$
- un insieme *più che numerabile* di **costanti** $\mathcal{Z}_{\mathcal{FDS}} \stackrel{def}{=} \mathcal{Z}_{\mathcal{FD}} \cup \mathcal{Z}_{\mathcal{FS}}$.

Definizione 7.8 (Termini di $\mathcal{L}_{\mathcal{FDS}}$)**.** L'insieme $\mathcal{T}_{\mathcal{FDS}}$ dei **termini** del linguaggio $\mathcal{L}_{\mathcal{FDS}}$ è definito come

$$\mathcal{T}_{\mathcal{FDS}} \stackrel{def}{=} \mathcal{T}^{\#}_{\mathcal{FD}} \cup \mathcal{T}_{\mathcal{FS}}$$

dove $\mathcal{T}^{\#}_{\mathcal{FD}} \stackrel{def}{=} \mathcal{T}_{\mathcal{FD}} \cup \{\#s : s \in \mathcal{T}_{\mathcal{FS}}\}$.

Definizione 7.9 (Vincoli atomici di $\mathcal{L}_{\mathcal{FDS}}$)**.** L'insieme $\mathcal{AC}_{\mathcal{FDS}}$ dei **vincoli atomici** del linguaggio $\mathcal{L}_{\mathcal{FDS}}$ è così definito:

(i) se $a \in \mathcal{AC}_{\mathcal{FS}}$ allora $a \in \mathcal{AC}_{\mathcal{FDS}}$

(ii) se $t_1, t_2 \in \mathcal{T}^{\#}_{\mathcal{FD}}$ e $* \in \mathcal{P}_{\mathcal{FD}}$ allora $t_1 * t_2 \in \mathcal{AC}_{\mathcal{FDS}}$

(iii) se $t \in \mathcal{T}^{\#}_{\mathcal{FD}}$ e $s \in \mathcal{T}_{\mathcal{FS}}$ allora $t \in s$ e $t \notin s$ appartengono a $\mathcal{AC}_{\mathcal{FDS}}$

(iv) nient'altro appartiene a $\mathcal{AC}_{\mathcal{FDS}}$.

Definizione 7.10 (Vincoli di $\mathcal{L}_{\mathcal{FDS}}$)**.** L'insieme $\mathcal{C}_{\mathcal{FDS}}$ dei **vincoli** del linguaggio $\mathcal{L}_{\mathcal{FDS}}$ è così definito:

(i) ogni vincolo atomico è un vincolo: $\mathcal{AC}_{\mathcal{FDS}} \subseteq \mathcal{C}_{\mathcal{FDS}}$

(ii) se $c_1, c_2 \in \mathcal{C}_{\mathcal{FDS}}$ allora $c_1 \wedge c_2 \in \mathcal{C}_{\mathcal{FDS}}$

(iii) nient'altro appartiene a $\mathcal{C}_{\mathcal{FDS}}$.

Esempio 9 (Termini e vincoli $\mathcal{L}_{\mathcal{FDS}}$). Come per $\mathcal{L}_{\mathcal{FD}}$ e $\mathcal{L}_{\mathcal{FS}}$, vediamo alcuni esempi di elementi che (non) appartengono agli insiemi sopracitati, utilizzando la stessa convenzione sulle parentesi dei linguaggi visti in precedenza.

Sono termini le seguenti espressioni:

- $X \cap \{1, 2, 3\}$
- $(1/y) \cdot (-2)$
- $\#(X \cup Y) - 1$

mentre non lo sono:

- $x \leq y + 1$
- $X = \#\{1, 3\}$

Sono invece vincoli le seguenti espressioni:

- $X :: [\{1\}..[1..4]] \wedge x :: \{1..10, 20..100\}$
- $\#(X \cup Y) \in Z \wedge 3 \notin Y$
- $X = \{1, 2\} \wedge Y \subset \{0, 5\} \wedge X \,||\, Y \wedge \ x \cdot x + y \cdot y = 1$

7.2.1 Semantica del linguaggio $\mathcal{L}_{\mathcal{FDS}}$

Per conferire significato ai simboli di $\mathcal{L}_{\mathcal{FDS}}$, utilizzeremo un'*interpretazione* Φ su $\mathbb{Z} \cup \mathcal{P}(\mathbb{Z})$ che associa:

- ad ogni simbolo $\delta \in \mathcal{P}_{\mathcal{FD}} \cup \mathcal{F}_{\mathcal{FD}} \cup \mathcal{Z}_{\mathcal{FD}}$ la corrispondente interpretazione $\delta^{\Phi} \stackrel{def}{=} \delta^{\Delta}$, dove Δ è l'interpretazione definita in 4.1.1
- ad ogni simbolo $\sigma \in \mathcal{P}_{\mathcal{FS}} \cup \mathcal{F}_{\mathcal{FS}} \cup \mathcal{Z}_{\mathcal{FS}}$ la corrispondente interpretazione $\delta^{\Phi} \stackrel{def}{=} \sigma^{\Sigma}$ dove Σ è l'interpretazione definita in 7.1.1
- ai predicati di (non) appartenenza insiemistica $\in$ e $\notin$ le relazioni $\in^{\Phi} \stackrel{def}{=} \in\restriction_{\mathbb{Z} \times \mathcal{P}(\mathbb{Z})}$ e $\notin^{\Phi} \stackrel{def}{=} \notin\restriction_{\mathbb{Z} \times \mathcal{P}(\mathbb{Z})}$
- al simbolo funzionale di *cardinalità insiemistica* $\#$ la funzione $\#^{\Phi} : \mathcal{P}(\mathbb{Z}) \longrightarrow \mathbb{Z}$ tale che, per ogni $X \in \mathcal{P}(\mathbb{Z})$, $\#^{\Phi}(X) = |X|$.

Fissata questa interpretazione, possiamo definire formalmente il concetto di *soddisfacimento di vincoli* in modo analogo a quanto fatto in precedenza per $\mathcal{L}_{\mathcal{FD}}$ e $\mathcal{L}_{\mathcal{FS}}$.

Definizione 7.11 (Funzione di assegnazione)**.** Siano $\mathbb{Z}_\alpha$ l'insieme universo definito in 3.2 e $\mathbb{Z}_\beta$ l'insieme universo definito in 6.2.

Si definisce (*funzione di*) **assegnazione** su $\mathcal{V}_{\mathcal{FDS}}$ una qualsiasi funzione $\varphi : \mathcal{V}_{\mathcal{FDS}} \longrightarrow \mathbb{Z}_\alpha \cup \mathcal{P}(\mathbb{Z}_\beta)$ tale che per ogni $v \in \mathcal{V}_{\mathcal{FDS}}$:

$$\varphi(v) \in \begin{cases} \mathbb{Z}_\alpha, & \text{se } v \in \mathcal{V}_{\mathcal{FD}} \\ \mathcal{P}(\mathbb{Z}_\beta) & \text{se } v \in \mathcal{V}_{\mathcal{FS}} \end{cases}$$

Una volta definita tale φ, è possibile ricavarne la sua **estensione ai termini** $\varphi^\Phi : \mathcal{T}_{\mathcal{FDS}} \longrightarrow \mathbb{Z} \cup \mathcal{P}(\mathbb{Z})$ tale che:

$$\begin{aligned} &\varphi^\Phi(t) \stackrel{def}{=} \varphi^\Delta(t), && \text{per ogni } t \in \mathcal{T}_{\mathcal{FD}} \\ &\varphi^\Phi(s) \stackrel{def}{=} \varphi^\Sigma(s), && \text{per ogni } s \in \mathcal{T}_{\mathcal{FS}} \\ &\varphi^\Phi(\#s) = |\varphi^\Sigma(s)|, && \text{per ogni } s \in \mathcal{T}_{\mathcal{FS}} \end{aligned}$$

Definizione 7.12 (Soddisfacimento di vincoli)**.** Sia $c \in \mathcal{C}_{\mathcal{FDS}}$ e φ una funzione di assegnazione su $\mathcal{V}_{\mathcal{FDS}}$. Allora,

$$\begin{aligned} &\models_\varphi t :: T && \stackrel{def}{\Longleftrightarrow} \varphi^\Phi(t) \in T, && t \in \mathcal{T}_{\mathcal{FDS}}, T \in (\mathbb{M}_\alpha \cup \mathbb{S}_\beta) \\ &\models_\varphi t_1 * t_2 && \stackrel{def}{\Longleftrightarrow} \varphi^\Phi(t_1) *^\Phi \varphi^\Phi(t_2), && t_1, t_2 \in \mathcal{T}_{\mathcal{FDS}}, * \in \mathcal{P}_{\mathcal{FDS}} \\ &\models_\varphi c_1 \wedge \ldots \wedge c_k && \stackrel{def}{\Longleftrightarrow} \underset{i=1,\ldots,k}{\forall} \models_\varphi c_i, && c_1, \ldots, c_k \in \mathcal{C}_{\mathcal{FDS}} \end{aligned}$$

Diremo quindi che un vincolo $c \in \mathcal{C}_{\mathcal{FDS}}$ (non) è **soddisfacibile** se (non) esiste un'assegnazione $\varphi : \mathcal{V}_{\mathcal{FDS}} \longrightarrow \mathbb{Z}_\alpha \cup \mathcal{P}(\mathbb{Z}_\beta)$ che lo soddisfa.

Esempio 10 (Soddisfacimento di vincoli). Consideriamo il vincolo:

$$X :: [\varnothing..[1..5]] \wedge \#X \notin X$$

Tale vincolo è soddisfacibile: si prenda ad esempio φ tale che $\varphi(X) = \{1, 3\}$.

Si ha che:

$$\begin{aligned}
&\models_\varphi X :: [\varnothing..[1..5]] \wedge \#X \notin X && \iff \\
&\models_\varphi X :: [\varnothing..[1..5]] \text{ e } \models_\varphi \#X \notin X && \iff \\
&\varphi^\Phi(X) \in [\varnothing..[1..5]] \text{ e } \varphi^\Phi(\#X) \notin \varphi^\Phi(X) && \iff \\
&\varphi(X) \in [\varnothing..[1..5]] \text{ e } |\varphi^\Sigma(X)| \notin \varphi(X) && \iff \\
&\{1,3\} \in [\varnothing..[1..5]] \text{ e } |\{1,3\}| \notin \{1,3\} && \iff \\
&\varnothing \subseteq \{1,3\} \subseteq [1..5] \text{ e } 2 \notin \{1,3\}
\end{aligned}$$

Infine, anche per $\mathcal{L}_{\mathcal{FDS}}$ le nozioni (e le rispettive notazioni) di **conseguenza** ed **equivalenza logica** di (insiemi di) vincoli appartenenti a $\mathcal{C}_{\mathcal{FDS}}$ sono esattamente le stesse date in 4.7 per i vincoli di $\mathcal{C}_{\mathcal{FD}}$.

7.3 Risoluzione dei vincoli di $\mathcal{L}_{\mathcal{FDS}}$

Occupiamoci finalmente di problemi di risoluzione di vincoli basati sul linguaggio $\mathcal{L}_{\mathcal{FDS}}$.

Definizione 7.13 ($CSP(\mathcal{FDS})$)**.** Un CSP basato su $\mathcal{L}_{\mathcal{FDS}}$, in breve $CSP(\mathcal{FDS})$, è un CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ nel quale:

- $\mathcal{V} = \{v_1, \ldots, v_n\} \subseteq \mathcal{V}_{\mathcal{FDS}}$ tale che, per ogni $X \in \mathcal{V}$,

$$X \in \mathcal{V}_{\mathcal{FS}} \iff (\exists\, cX \in \mathcal{V} \cap \mathcal{V}_{\mathcal{FD}})\ cX = \#X$$

- $\mathcal{D} = D_1 \times \ldots \times D_n$ dove per $i = 1, \ldots, n$ si ha che $v_i :: D_i$, quindi

$$D_i \in \begin{cases} \mathbb{M}_\alpha, & \text{se } v_i \in \mathcal{V}_{\mathcal{FD}} \\ \mathbb{S}_\beta & \text{se } v_i \in \mathcal{V}_{\mathcal{FS}} \end{cases}$$

- $\mathcal{C} = \{c_1, \ldots, c_m\} \subseteq \mathcal{C}_{\mathcal{FDS}}$

L'insieme $\mathcal{C}$ è detto anche **constraint store** di $\mathcal{P}$

Anzitutto, chiariamo la definizione di $\mathcal{V}$. La doppia implicazione introdotta sta a significare che per ogni variabile *insiemistica* $X \in \mathcal{V}$ esiste una corrispondente variabile *intera* $cX \in \mathcal{V}$ che di fatto rappresenta la **cardinalità** di X.

Questo perché, come si noterà meglio in seguito, l'informazione aggiuntiva introdotta dalla cardinalità insiemistica sarà cruciale nella risoluzione di vincoli di $\mathcal{L}_{\mathcal{FDS}}$.

Ad esempio, se una variabile X ha dominio $[\varnothing..[1..3]]$ e la cardinalità associata è $cX = 3$, non potrà che essere $X = [1..3]$.
Se invece fosse $cX = 1$, avremmo che $X = \{1\} \vee X = \{2\} \vee X = \{3\}$.

Per alleggerire la notazione, d'ora in avanti le variabili di $\mathcal{V}$ verranno indicate con le meta-variabili minuscole $x, y, z, \ldots$ nel caso siano variabili $\mathcal{FD}$ mentre si utilizzerà la notazione maiuscola $X, Y, Z, \ldots$ per indicare le variabili $\mathcal{FS}$.
I corrispondenti domini saranno denotati con $D_x, D_y, D_z, \ldots, D_X, D_Y, D_Z, \ldots$

Definizione 7.14 (Constraint store semplificato)**.** Sia $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ un $CSP(\mathcal{FDS})$. Dal constraint store $\mathcal{C}$ è possibile ottenere un insieme di vincoli atomici *logicamente equivalente* $\mathcal{CS} \subseteq \mathcal{AC}_{\mathcal{FDS}}$ tale che ogni vincolo $a \in \mathcal{CS}$ è della forma:

(i) descritta in 4.9, se a contiene esclusivamente termini appartenenti a $\mathcal{T}^{\#}_{\mathcal{FD}}$

(ii) $x \in X$ oppure $x \notin X$

(iii) $X :: S$

(iv) $X * Y$ oppure $X * A, \quad$ se $* \in \mathcal{P}_{\mathcal{FS}} \setminus \{\subset\}$

(v) $Z = \sim X$ oppure $Z = X * Y, \quad$ se $* \in \mathcal{F}_{\mathcal{FS}}$.

dove $x \in \mathcal{V}_{\mathcal{FD}}$, $X, Y, Z \in \mathcal{V}_{\mathcal{FS}}$, $A \in \mathbb{M}_\alpha$ e $S \in \mathbb{S}_\beta$.
L'insieme $\mathcal{CS}$ verrà detto **constraint store semplificato** di $\mathcal{P}$.

Osservazione 11 (Semplificazione del constraint store). Come per 4.9, osserviamo come sia sempre possibile semplificare $\mathcal{C}$ in un constraint store logicamente equivalente $\mathcal{CS}$, infatti:

- ogni vincolo che soddisfa la pre-condizione di **(i)** viene semplificato secondo l'approccio definito in 7
- ogni vincolo non atomico della forma $c_1 \wedge \ldots \wedge c_k$ viene 'espanso' fino ad ottenere soli vincoli atomici logicamente equivalenti $a_1, \ldots, a_h$, con $h \geq k$
- il vincolo $\subset$ viene riscritto in forma di $\subseteq$, in quanto

$$X \subset Y \models\!\!\models \{X \subseteq Y, \#X < \#Y\}$$

- ogni altro vincolo può essere riportato nelle forme **(ii)**, ..., **(iv)** eventualmente aggiungendo nuovi vincoli e variabili ausiliarie.

Teorema 7.3.1 (Consistenza e soddisfacibilità)**.** *Sia* $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ *un* $CSP(\mathcal{FDS})$ *e* $\mathcal{CS}$ *il corrispondente constraint store semplificato. Allora,*

$$\mathcal{P} \textit{ consistente} \iff \mathcal{CS} \textit{ soddisfacibile}$$

Dimostrazione. Identica alla dimostrazione del teorema 4.2.1. ■

Come in 4.2.1, dalla proposizione segue che un $CSP(\mathcal{FDS})$ ha (almeno) una soluzione se e soltanto se il corrispondente $\mathcal{CS}$ è soddisfacibile, cioè esiste un'assegnazione di valori alle variabili di $\mathcal{CS}$ che soddisfa ogni suo vincolo atomico.

Come per i vincoli di $\mathcal{C}_{\mathcal{FD}}$, per risolvere i vincoli di $\mathcal{C}_{\mathcal{FDS}}$ verranno utilizzate apposite *regole di riduzione dei domini*.
Tuttavia, considerata la maggiore complessità del linguaggio $\mathcal{L}_{\mathcal{FDS}}$ rispetto a $\mathcal{L}_{\mathcal{FD}}$, tali regole verranno formalizzate in modo differente rispetto all'approccio seguito in 4.3.

Definizione 7.15 (Regola di riduzione)**.** Sia $a \in \mathcal{AC}_{\mathcal{FDS}}$ un vincolo atomico di $\mathcal{L}_{\mathcal{FDS}}$.
Una **regola di riduzione** dei domini per a è un'operazione R_a che, nel caso siano soddisfatte un certo numero di *pre-condizioni* $P_1, \ldots, P_h$, permette *eventualmente* di:

- *modificare* il constraint store corrente $\mathcal{CS}$ in un nuovo constraint store $\mathcal{CS}'$
- *restringere* un certo numero di domini $D_{i_1}, \ldots, D_{i_k}$ sul quale a è definito, calcolando nuovi domini $D'_{i_j} \subseteq D_{i_j}$ ottenuti intersecando D_{i_j} con un opportuno insieme di valori Θ_{i_j} per $j = 1, \ldots, k$.

Per indicare una tale regola useremo la notazione:

$$R_a : \frac{P_1, \ldots, P_h}{\mathcal{CS} \mapsto \mathcal{CS}' \quad D'_{i_1} = D_{i_1} \cap \Theta_{i_1} \cdots D'_{i_k} = D_{i_k} \cap \Theta_{i_k}}$$

Si noti anzitutto che per i vincoli atomici della forma **(i)** di 7.14 si utilizzeranno le regole di riduzione definite in 4.3, essendo a tutti gli effetti vincoli su interi.
Nella prossima sezione verranno quindi presentate esclusivamente le regole di riduzione dei vincoli la cui forma ricade nei casi **(ii)**, ..., **(v)** di 7.14.

L'approccio di *constraint solving* sarà lo stesso definito per i vincoli di $\mathcal{L}_{\mathcal{FD}}$: dapprima viene semplificato il constraint store, quindi viene iterata in modo sequenziale la risoluzione locale dei suoi vincoli fino al raggiungimento di un punto fisso (come si vedrà in seguito, ciò è garantito in un *numero finito* di iterazioni).

7.4 Regole di riduzione dei domini

In questa sezione verranno presentate le regole di riduzione dei domini per ciascun vincolo atomico del constraint store $\mathcal{CS}$ che ricade nei casi **(ii)**, ..., **(v)** di 7.14.

Prima però conviene notare che quasi tutte queste regole *non* rispettano in generale la nozione di arc-consistency: ciò risulterebbe decisamente troppo costoso, considerando il maggior costo delle operazioni insiemistiche rispetto a quelle fra interi e il numero esponenziale degli elementi appartenenti ad un set-interval.

In generale invece verrà rispettata la nozione di **bound-consistency** (in altri termini, l'arc-consistency viene mantenuta solamente sui *bounds* delle variabili).

L'approccio di risoluzione seguirà quanto descritto in [3].

7.4.1 Vincoli di dominio e cardinalità

Quando un vincolo di dominio del tipo $X :: S$ viene aggiunto al constraint store $\mathcal{CS}$, in esso verrà aggiunto anche il vincolo $cX :: [|S^-|..|S^+|]$.
Ciò impone che la cardinalità di X sia compresa fra le cardinalità degli estremi del dominio S.

$$R_{::} : \frac{add(X :: S)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{X :: S,\ cX :: [|S^-|..|S^+|]\}} \tag{7.1}$$

I vincoli di dominio della forma $X :: S$ vengono risolti semplicemente intersecando il dominio corrente D_X con il set-interval S.
Una volta aggiornato D_X in D'_X, il vincolo può essere rimosso dallo store.

$$R_{::} : \frac{X :: S}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X :: S\} \quad D'_X = D_X \cap S} \tag{7.2}$$

7.4.2 Vincolo di cardinalità

Per risolvere vincoli di cardinalità del tipo $n = \#X$ è in realtà sufficiente imporre $n = cX$ e successivamente gestire la variabile di cardinalità cX in base ai vincoli insiemistici che coinvolgono X.

$$R_{\#} : \frac{add(n = \#X)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{n = cX\}} \tag{7.3}$$

Quando il dominio di una variabile insiemistica X viene aggiornato in D'_X, la sua cardinalità va modificata di conseguenza: essa dovrà essere compresa

fra $|D'^{-}_X|$ e $|D'^{+}_X|$.

$$R_{\#} : \frac{update(D_X \mapsto D'_X)}{D'_{cX} = D_{cX} \cap [|D'^{-}_X|..|D'^{+}_X|]} \tag{7.4}$$

Inoltre, se cX corrisponde alla cardinalità di uno dei bound di D_X, allora X assumerà il valore di tale bound.

$$R_{\#} : \frac{cX = |D^{-}_X|}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{X = D^{-}_X\}} \tag{7.5}$$

$$R_{\#} : \frac{cX = |D^{+}_X|}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{X = D^{+}_X\}} \tag{7.6}$$

Ad esempio, sia $D_X = [\varnothing..[1..5]]$. Se $cX = |\varnothing| = 0$, allora sarà $X = \varnothing$; se invece $cX = |[1..5]| = 5$, allora sarà $X = [1..5]$.

7.4.3 Vincoli di (non) appartenenza

I vincoli di (non) appartenenza insiemistica comportano una riduzione dei domini solamente nel caso la variabile intera x coinvolta sia *known*: ciò significa che il suo valore è noto, essendo il suo dominio ridotto ad un singoletto.
Prima di descrivere le regole, conviene osservare che per ogni variabile insiemistica X si ha che:

- D^{-}_X è l'insieme degli elementi che *sicuramente appartengono* ad X
- D^{+}_X è l'insieme degli elementi che *potrebbero appartenere* ad X.

Nel caso dell'appartenenza quindi il valore di x viene aggiunto al GLB della corrispondente variabile insiemistica X:

$$R_{\in} : \frac{add(x \in X)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{x \in X\}} \tag{7.7}$$

$$R_{\in} : \frac{x \in X,\ D_x = \{k\},\ A = D^{-}_X \cup \{k\}}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{x \in X\} \quad D'_X = D_X \cap [A..D^{+}_X]} \tag{7.8}$$

Viceversa, se il vincolo è di non appartenenza il valore di x viene rimosso dal LUB di X.

$$R_{\notin} : \frac{add(x \notin X)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{x \notin X\}} \tag{7.9}$$

$$R_{\notin} : \frac{x \notin X,\ D_x = \{k\},\ B = D^{+}_X \setminus \{k\}}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{x \notin X\} \quad D'_X = D_X \cap [D^{-}_X..B]} \tag{7.10}$$

In entrambi i casi 7.8 e 7.10 il vincolo può considerarsi risolto, e quindi rimosso dal constraint store.

7.4.4 Vincolo di uguaglianza

Se due insiemi X e Y sono uguali, allora hanno la stessa cardinalità: per questa ragione quando si inserisce nel constraint store un vincolo del tipo $X = Y$ viene aggiunto anche il vincolo $cX = cY$.

$$R_{=} : \frac{add(X = Y)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{X = Y,\ cX = cY\}} \tag{7.11}$$

Inoltre, se due insiemi sono uguali devono avere lo stesso dominio:

$$R_{=} : \frac{X = Y,\ D = D_X \cap D_Y}{D'_X = D \quad D'_Y = D} \tag{7.12}$$

Nel caso in cui il secondo termine dell'uguaglianza sia una *costante* insiemistica è possibile rimuovere il vincolo dallo store (se $A \in D_X$ si avrà che $D'_X = \{A\}$; viceversa si verificherà un fallimento essendo $D'_X = \varnothing$).

$$R_{=} : \frac{X = A,\ A \in \mathbb{M}_\alpha}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X = A\} \quad D'_X = D_X \cap \{A\}} \tag{7.13}$$

Infine, le precedenti regole consentono di rimuovere il vincolo $X = Y$ dal constraint store qualora una fra X e Y diventi *known*.

$$R_{=} : \frac{X = Y,\ (known(X) \vee known(Y))}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X = Y\}} \tag{7.14}$$

7.4.5 Vincolo di disuguaglianza

Asimmetricamente rispetto all'uguaglianza insiemistica, il fatto che due insiemi X e Y siano diversi non implica necessariamente che siano diverse anche le loro cardinalità (si pensi banalmente agli insiemi $\{0\}$ e $\{1\}$).

$$R_{\neq} : \frac{add(X \neq Y)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{X \neq Y\}} \tag{7.15}$$

Il vincolo di diverso non permette in generale di effettuare molta riduzione; tuttavia, è possibile verificarne la consistenza nei casi 'degeneri':

$$R_{\neq} : \frac{X \neq Y,\ X = Y}{\mathcal{CS} \mapsto fail} \tag{7.16}$$

$$R_{\neq} : \frac{X \neq Y,\ D_X = \{A\},\ D_Y = \{B\},\ A \neq B}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \neq Y\}} \tag{7.17}$$

$$R_{\neq} : \frac{X \neq A,\ A \in \mathbb{M}_{\alpha},\ X = A}{\mathcal{CS} \mapsto fail} \tag{7.18}$$

$$R_{\neq} : \frac{X \neq A,\ A \in \mathbb{M}_{\alpha}, D_X = \{B\},\ A \neq B}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \neq Y\}} \tag{7.19}$$

Si noti che 7.17 e 7.19 potrebbero essere rispettivamente sostituite dalle più generiche regole:

$$R_{\neq} : \frac{X \neq Y,\ D_X \cap D_Y = \varnothing}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \neq Y\}}$$

$$R_{\neq} : \frac{X \neq A,\ A \in \mathbb{M}_{\alpha},\ A \notin D_X}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \neq Y\}}$$

La loro effettiva utilità tuttavia non compensa il maggior costo computazionale; infatti, tali regole non comportano una riduzione dei domini ma solamente la rimozione del vincolo dallo store.
Si è quindi deciso di mantenere 7.17 e 7.19 posticipando la risoluzione al momento dell'eventuale istanziazione delle variabili coinvolte.

7.4.6 Vincolo di disgiunzione

Qualora due insiemi X e Y siano disgiunti, cioè $X \cap Y = \varnothing$, la somma delle loro cardinalità non può eccedere la cardinalità dell'universo $\mathbb{Z}_{\beta}$, infatti:

$$|X| + |Y| = |X \cup Y| - |X \cap Y| = |X \cup Y| \leq |\mathbb{Z}_{\beta}|$$

$$R_{||} : \frac{add(X \,||\, Y)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{X \,||\, Y,\ cX + cY \leq |\mathbb{Z}_{\beta}|\}} \tag{7.20}$$

Gli elementi che sicuramente appartengono ad un insieme, non possono ovviamente appartenere all'altro:

$$R_{||} : \frac{X \,||\, Y,\ A = D_X^+ \setminus D_Y^-,\ B = D_Y^+ \setminus D_X^-}{D'_X = D_X \cap [D_X^- .. A] \quad D'_Y = D_Y \cap [D_Y^- .. B]} \tag{7.21}$$

Nel caso particolare in cui $X = Y$, allora dev'essere per forza $X = Y = \varnothing$: il vincolo $X \,||\, Y$ può essere rimosso dallo store:

$$R_{||} : \frac{X \,||\, Y,\ X = Y}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{X \,||\, Y\}) \cup \{X = \varnothing\}} \tag{7.22}$$

Va gestito inoltre il caso in cui compaia una costante insiemistica nel vincolo:

$$R_{||} : \frac{X \,||\, A,\ A \in \mathbb{M}_{\alpha}}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \,||\, A\} \quad D'_X = D_X \cap [D_X^- .. D_X^+ \setminus A]} \tag{7.23}$$

Come per l'uguaglianza, le precedenti consentono di rimuovere il vincolo $X \,||\, Y$ dal constraint store qualora una fra X e Y diventi *known*.

$$R_{||} : \frac{X \,||\, Y,\ (known(X) \vee known(Y))}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \,||\, Y\}} \tag{7.24}$$

7.4.7 Vincolo di inclusione

Se un insieme X è contenuto in un altro insieme Y, allora la sua cardinalità non potrà superare quella di Y:

$$R_{\subseteq} : \frac{add(X \subseteq Y)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{X \subseteq Y,\ cX \leq cY\}} \tag{7.25}$$

In particolare, se le cardinalità dei due insiemi coincidono allora di fatto sono lo stesso insieme:

$$R_{\subseteq} : \frac{X \subseteq Y,\ cX = cY}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{X \subseteq Y\}) \cup \{X = Y\}} \tag{7.26}$$

Se ogni valore che potrebbe appartenere ad X appartiene sicuramente ad Y, allora il vincolo è risolto:

$$R_{\subseteq} : \frac{X \subseteq Y,\ D_X^+ \subseteq D_Y^-}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \subseteq Y\}} \tag{7.27}$$

Inoltre, ogni valore che potrebbe appartenere ad Y potrebbe appartenere anche ad X; simmetricamente se un valore appartiene sicuramente ad X allora appartiene anche ad Y.

$$R_{\subseteq} : \frac{X \subseteq Y,\ A = D_X^- \cup D_Y^-,\ B = D_X^+ \cap D_Y^+}{D'_X = D_X \cap [D_X^-..B] \quad D'_Y = D_Y \cap [A..D_Y^+]} \tag{7.28}$$

Se una costante compare nel vincolo, posso ridurre il dominio della variabile insiemistica coinvolta e togliere il vincolo dallo store.

$$R_{\subseteq} : \frac{X \subseteq A,\ A \in \mathbb{M}_\alpha, B = D_X^+ \cap A}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \subseteq A\} \quad D'_X = D_X \cap [D_X^-..B]} \tag{7.29}$$

A questo punto, se una fra X e Y diventa *known*, posso rimuovere il vincolo dal constraint store.

$$R_{\subseteq} : \frac{X \subseteq Y,\ (known(X) \vee known(Y))}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{X \subseteq Y\}} \tag{7.30}$$

7.4.8 Vincolo di complementazione

Quando il vincolo $Z = \sim X$ viene aggiunto al constraint store, impongo che la somma della cardinalità dei due insiemi sia esattamente la cardinalità dell'universo $\mathbb{Z}_\beta$, in quanto $X \cup Z = \mathbb{Z}_\beta$ e $X \cap Z = \varnothing$ perciò:

$$|X| + |Z| = |X \cup Z| - |X \cap Z| = |\mathbb{Z}_\beta| - |\varnothing| = |\mathbb{Z}_\beta|$$

$$R_\sim : \frac{add(Z = \sim X)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{Z = \sim X,\ cX + cZ = |\mathbb{Z}_\beta|\}} \tag{7.31}$$

Nel caso in cui i due insiemi siano uguali, allora l'universo non può che essere l'insieme vuoto. Nel nostro caso, essendo $\mathbb{Z}_\beta = [-\beta..\beta] \neq \varnothing$, si avrà quindi fallimento della risoluzione:

$$R_\sim : \frac{Z = \sim X,\ Z = X}{\mathcal{CS} \mapsto fail} \tag{7.32}$$

Se X e Z sono *known*, allora il vincolo è banalmente risolto confrontando i valori dei due insiemi.

$$R_\sim : \frac{Z = \sim X,\ D_X = \{A\},\ D_Z = \{C\},\ C = \sim A}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{Z = \sim X\}} \tag{7.33}$$

$$R_\sim : \frac{Z = \sim X,\ D_X = \{A\},\ D_Z = \{C\},\ C \neq \sim A}{\mathcal{CS} \mapsto fail} \tag{7.34}$$

Nel caso in cui le variabili non risultino istanziate si applica una riduzione dei domini: gli elementi che *non possono appartenere* ad un insieme staranno sicuramente nell'altro, mentre gli elementi che *potrebbero non appartenere* ad un insieme potrebbero invece ricadere nell'altro.

$$R_\sim : \frac{Z = \sim X}{\begin{array}{l} D'_X = D_X \cap [\sim(D_Z^+) \,..\, \sim(D_Z^-)] \\ D'_Z = D_Z \cap [\sim(D_X^+) \,..\, \sim(D_X^-)] \end{array}} \tag{7.35}$$

7.4.9 Vincolo di intersezione

Il vincolo $Z = X \cap Y$ implica che $Z \subseteq X$ e $Z \subseteq Y$. Tuttavia, sempre seguendo l'approccio definito in [3], quando il vincolo viene aggiunto allo store un altro speciale vincolo di cardinalità $cZ :: (X \cap_\# Y)$ viene imposto:

$$R_\cap : \frac{add(Z = X \cap Y)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \{Z = X \cap Y,\ add(Z \subseteq X),\ add(Z \subseteq Y),\ cZ :: (X \cap_\# Y)\}} \tag{7.36}$$

Formalmente, se cX e cY sono costituiti dai seguenti intervalli:

$$cX = \{X_1, \ldots, X_n\} \quad cY = \{Y_1, \ldots, Y_m\}$$

allora per $i = 1, \ldots, n$ e $j = 1, \ldots, m$ il multi-intervallo $X \cap_{\#} Y$ è definito dalla seguente unione:

$$X \cap_{\#} Y \stackrel{def}{=} \bigcup [X_i^- + Y_j^- - u \mathinner{..} min(X_i^+ - a, Y_j^+ - b)]$$

con $a = |D_X^- \setminus D_Y^+|$, $b = |D_Y^- \setminus D_X^+|$ e $u = |D_X^+ \cup D_Y^+|$.

Data l'evidente complessità di questa formulazione, non ci si inoltrerà nella sua descrizione formale. Si noti invece che l'aggiunta di questo vincolo permette di mantenere l'arc-consistency su cX, cY e cZ.

Siano ad esempio X, Y e Z tali che:

$$D_X = D_Y = [\varnothing..\{0, 1\}] \quad cX = cY = \{0, 2\} \quad Z = X \cap Y.$$

Senza il vincolo $cZ :: (X \cap_{\#} Y)$ si avrebbe che $cZ :: [0..2]$ quando in realtà $cZ :: \{0, 2\}$. Invece, essendo

$$X \cap_{\#} Y = [-2..0] \cup [0..0] \cup [0..0] \cup [2..2] = \{-2..0, 2\}$$

risulta giustamente che $cZ :: [0..2] \cap \{-2..0, 2\} = \{0, 2\}$. Questa maggiore precisione risulta molto utile in problemi reali come l'analisi di *circuiti digitali*.[3]

Si noti tuttavia che $X \cap_{\#} Y$ viene calcolato solamente quando $Z = X \cap Y$ viene inserito nello store: in seguito, solo la bound-consistency è mantenuta sui vincoli di cardinalità.

Se $X = Y$, allora il vincolo è risolto: si avrà che $Z = X$.

$$R_{\cap} : \frac{Z = X \cap Y,\ X = Y}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cap Y\}) \cup \{Z = X\}} \tag{7.37}$$

Analogamente, se $X = Z \vee Y = Z$ posso rimuovere il vincolo dallo store: infatti, ciò implica che $Z \subseteq Y \vee Z \subseteq X$ (entrambi questi vincoli sono già presenti in esso, vedi regola 7.36).

$$R_{\cap} : \frac{Z = X \cap Y,\ (X = Z \vee Y = Z)}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{Z = X \cap Y\}} \tag{7.38}$$

Se X ed Y sono *known*, allora rimuovo il vincolo e pongo semplicemente Z uguale all'intersezione dei valori di X e Y.

$$R_{\cap} : \frac{Z = X \cap Y,\ X = \{A\},\ Y = \{B\}}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cap Y\}) \cup \{Z = A \cap B\}} \tag{7.39}$$

Se $D_X^+ \subseteq D_Y^-$, allora $D_X \subseteq D_Y$ da cui segue che $Z = X$.

$$R_\cap : \frac{Z = X \cap Y,\ D_X^+ \subseteq D_Y^-}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cap Y\}) \cup \{Z = X\}} \tag{7.40}$$

Analogamente, se $D_Y^+ \subseteq D_X^-$ allora $Z = Y$.

$$R_\cap : \frac{Z = X \cap Y,\ D_Y^+ \subseteq D_X^-}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cap Y\}) \cup \{Z = Y\}} \tag{7.41}$$

Inoltre, Z conterrà tutti i valori che appartengono ad X e ad Y:

$$R_\cap : \frac{Z = X \cap Y,\ A = D_X^- \cap D_Y^-}{D'_Z = D_Z \cap [A..D_Z^+]} \tag{7.42}$$

Per definizione di intersezione, tutti e soli gli elementi che appartengono sia ad X che ad Y devono stare in Z.
Quindi, tra i possibili elementi di X vanno rimossi quelli che appartengono ad Y ma non possono appartenere a Z.

$$R_\cap : \frac{Z = X \cap Y,\ A = D_X^+ \setminus (D_Y^- \setminus D_Z^+)}{D'_X = D_X \cap [D_X^-..A]}$$

Simmetricamente, tra i possibili elementi di Y vanno rimossi quelli che appartengono ad X ma possono non appartenere a Z.

$$R_\cap : \frac{Z = X \cap Y,\ B = D_Y^+ \setminus (D_X^- \setminus D_Z^+)}{D'_Y = D_Y \cap [D_Y^-..B]}$$

Come si può notare, queste regole sono molto più costose delle precedenti: esse contengono infatti una *doppia operazione* di differenza insiemistica.
Per questo motivo, si è scelto di applicarle solamente quando le variabili sono sufficientemente istanziate:

$$R_\cap : \frac{Z = X \cap Y,\ Y = \{B\},\ Z = \{C\},\ A = D_X^+ \setminus (B \setminus C)}{D'_X = D_X \cap [D_X^-..A]} \tag{7.43}$$

$$R_\cap : \frac{Z = X \cap Y,\ X = \{A\},\ Z = \{C\},\ B = D_Y^+ \setminus (A \setminus C)}{D'_Y = D_Y \cap [D_Y^-..B]} \tag{7.44}$$

7.4.10 Vincolo di unione

Come per l'intersezione, anche l'inserimento del vincolo di unione al constraint store implica l'aggiunta di nuovi vincoli:

$$R_\cup : \frac{add(Z = X \cup Y)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \left\{ \begin{array}{c} Z = X \cup Y,\ add(X \subseteq Z),\ add(Y \subseteq Z), \\ cZ \leq cX + cY, \quad cZ :: (X \cup_\# Y) \end{array} \right\}} \tag{7.45}$$

Infatti, se $Z = X \cup Y$ allora si ha che $X \subseteq Z$, $Y \subseteq Z$ e $cZ \leq cX + cY$. Inoltre, analogamente a quanto fatto per l'intersezione, il dominio di cZ viene ulteriormente raffinato mediante l'aggiunta del vincolo $cZ :: (X \cup_{\#} Y)$. Formalmente, se cX e cY sono costituiti dai seguenti intervalli:

$$cX = \{X_1, \ldots, X_n\} \quad cY = \{Y_1, \ldots, Y_m\}$$

allora per $i = 1, \ldots, n$ e $j = 1, \ldots, m$ il multi-intervallo $X \cup_{\#} Y$ è definito dalla seguente unione:

$$X \cup_{\#} Y \stackrel{def}{=} \bigcup [max(X_i^- + a,\ Y_j^- + b) \mathinner{..} min(X_i^+ + Y_j^+,\ u)]$$

con $a = |D_Y^- \setminus D_X^+|$, $b = |D_X^- \setminus D_Y^+|$ e $u = |D_X^+ \cup D_Y^+|$.

Per comprendere l'utilità di questo ulteriore vincolo, si consideri il seguente esempio.

Siano X, Y e Z tali che:

$$D_X = D_Y = [\varnothing..[0..3]] \quad cX = cY = \{0, 4\} \quad Z = X \cap Y.$$

Senza il vincolo $cZ :: (X \cup_{\#} Y)$ si avrebbe che $cZ :: [0..4]$ quando in realtà $cZ :: \{0, 4\}$. Invece, essendo

$$X \cup_{\#} Y = [0..0] \cup [4..0] \cup [4..0] \cup [4, 4] = \{0, 4\}$$

risulta giustamente che $cZ :: [0..4] \cap \{0, 4\} = \{0, 4\}$.

Se $X = Y$, il vincolo è risolto: si avrà che $Z = X$.

$$R_{\cup} : \frac{Z = X \cup Y,\ X = Y}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cup Y\}) \cup \{Z = X\}} \tag{7.46}$$

Analogamente, se $X = Z \vee Y = Z$ posso rimuovere il vincolo allo store senza effettuare ulteriore propagazione.

$$R_{\cup} : \frac{Z = X \cup Y,\ (X = Z \vee Y = Z)}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{Z = X \cup Y\}} \tag{7.47}$$

Se X e Y sono *known*, rimuovo il vincolo ponendo semplicemente Z uguale all'unione dei valori di X e Y.

$$R_{\cup} : \frac{Z = X \cup Y,\ X = \{A\},\ Y = \{B\}}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cup Y\}) \cup \{Z = A \cup B\}} \tag{7.48}$$

Se $D_X^+ \subseteq D_Y^-$, allora $D_X \subseteq D_Y$ da cui segue che $Z = Y$.

$$R_\cup : \frac{Z = X \cup Y,\ D_X^+ \subseteq D_Y^-}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cup Y\}) \cup \{Z = Y\}} \tag{7.49}$$

Analogamente, se $D_Y^+ \subseteq D_X^-$ allora $Z = X$.

$$R_\cup : \frac{Z = X \cup Y,\ D_Y^+ \subseteq D_X^-}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \cup Y\}) \cup \{Z = X\}} \tag{7.50}$$

Inoltre, Z potrà contenere tutti i valori che appartengono ad X e ad Y:

$$R_\cup : \frac{Z = X \cup Y,\ B = D_X^+ \cup D_Y^+}{D'_Z = D_Z \cap [D_Z^- .. B]} \tag{7.51}$$

Per definizione di unione, tutti e soli gli elementi che appartengono ad X oppure ad Y devono stare in Z.
Quindi, tra gli elementi che sicuramente appartengono ad X aggiungo quelli che stanno in Z ma non possono appartenere a Y.

$$R_\cup : \frac{Z = X \cup Y,\ A = D_X^- \cup (D_Z^- \setminus D_Y^+)}{D'_X = D_X \cap [A .. D_X^+]}$$

Simmetricamente, tra gli elementi che sicuramente appartengono ad Y aggiungo quelli che stanno in Z ma non possono appartenere a X.

$$R_\cup : \frac{Z = X \cup Y,\ B = D_Y^- \cup (D_Z^- \setminus D_Z^+)}{D'_Y = D_Y \cap [B .. D_Y^+]}$$

Come per l'intersezione tuttavia queste operazioni risultano costose, quindi vengono applicate solamente nel caso di una sufficiente istanziazione delle variabili:

$$R_\cup : \frac{Z = X \cup Y,\ Y = \{B\},\ Z = \{C\},\ A = D_X^- \cup (C \setminus B)}{D'_X = D_X \cap [A .. D_X^+]} \tag{7.52}$$

$$R_\cup : \frac{Z = X \cup Y,\ X = \{A\},\ Z = \{C\},\ B = D_Y^- \cup (C \setminus A)}{D'_Y = D_Y \cap [B .. D_Y^+]} \tag{7.53}$$

7.4.11 Vincolo di differenza insiemistica

Occupiamoci infine del vincolo di differenza insiemistica. Se $Z = X \setminus Y$, allora sicuramente $Z \subseteq X$, $Y \,||\, Z$ e $cZ \leq cX - cY$, in quanto:

$$\begin{aligned} |Z| &= |X \setminus Y| = |X \cap {\sim}Y| = |X| + |{\sim}Y| - |X \cup {\sim}Y| \\ &= |X| - |Y| + |\mathbb{Z}_\beta| - |X \cup {\sim}Y| \geq |X| - |Y| \end{aligned}$$

Inoltre, come per intersezione ed unione, verrà aggiunto uno speciale vincolo di dominio $cZ :: (cX \setminus_{\#} cY)$.

$$R_{\setminus} : \frac{add(Z = X \setminus Y)}{\mathcal{CS} \mapsto \mathcal{CS} \cup \left\{ \begin{array}{c} Z = X \setminus Y,\ add(Z \subseteq X),\ add(Y \,||\, Z), \\ cZ \leq cX - cY, \quad cZ :: (cX \setminus_{\#} cY) \end{array} \right\}} \tag{7.54}$$

Formalmente, se cX e cY sono costituiti dai seguenti intervalli:

$$cX = \{X_1, \ldots, X_n\} \quad cY = \{Y_1, \ldots, Y_m\}$$

allora per $i = 1, \ldots, n$ e $j = 1, \ldots, m$ il multi-intervallo $X \setminus_{\#} Y$ è definito dalla seguente unione:

$$X \setminus_{\#} Y \stackrel{def}{=} \bigcup [max(X_i^- - Y_j^-, X_i^- - a) \,..\, min(X_i^+ - b, u - Y_j^+)]$$

con $a = |D_Y^- \setminus D_X^+|$, $b = |D_X^- \setminus D_Y^+|$ e $u = |D_X^+ \cup D_Y^+|$.

Consideriamo ad esempio X, Y e Z tali che:

$$D_X = D_Y = [\varnothing..[0..3]] \quad cX = cY = \{0, 4\} \quad Z = X \cap Y.$$

Senza il vincolo $cZ :: (X \setminus_{\#} Y)$ si avrebbe che $cZ :: [0..4]$ quando in realtà $cZ :: \{0, 4\}$. Invece, essendo

$$X \setminus_{\#} Y = [0..0] \cup [0..0] \cup [4..4] \cup [4, 4] = \{0, 4\}$$

risulta giustamente che $cZ :: [0..4] \cap \{0, 4\} = \{0, 4\}$.

Si considerino invece $D_X = [\varnothing..[1..20]]$, $cX = 10$, $D_Y = [\varnothing..[20..30]]$. Senza il vincolo $cZ :: (X \setminus_{\#} Y)$, da $X \setminus Y$ si inferirebbe solamente che $cZ :: [1..10]$; grazie ad esso invece si ottiene $cZ :: [9..10]$.

Prendiamo ora $D_X = D_Y = [[1..5]..[1..20]]$ con $cX = cY = 6$.
Il vincolo $cZ :: (X \setminus_{\#} Y)$ permette di inferire che $cZ :: [0..1]$, mentre senza di esso risulterebbe $cZ :: [0..6]$.

Se $X = Y$ il vincolo è risolto: si avrà che $Z = \varnothing$.

$$R_{\setminus} : \frac{Z = X \setminus Y,\ X = Y}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \setminus Y\}) \cup \{Z = \varnothing\}} \tag{7.55}$$

Se $X = Z$, posso rimuovere il vincolo dallo store: X e Y sono disgiunti.

$$R_{\setminus} : \frac{Z = X \setminus Y,\ X = Z}{\mathcal{CS} \mapsto \mathcal{CS} \setminus \{Z = X \setminus Y\}} \tag{7.56}$$

Se X e Y sono *known*, rimuovo il vincolo ponendo semplicemente Z uguale alla differenza insiemistica dei valori di X e Y.

$$R_{\setminus} : \frac{Z = X \setminus Y,\ X = \{A\},\ Y = \{B\}}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \setminus Y\}) \cup \{Z = A \setminus B\}} \tag{7.57}$$

Se $D_X^+ \,||\, D_Y^+$, allora $X \,||\, Y$: rimuovo il vincolo dallo store e impongo $Z = Y$.

$$R_{\setminus} : \frac{Z = X \setminus Y,\ D_X^+ \,||\, D_Y^+}{\mathcal{CS} \mapsto (\mathcal{CS} \setminus \{Z = X \setminus Y\}) \cup \{Z = Y\}} \tag{7.58}$$

L'insieme Z conterrà tutti gli elementi che stanno in X ma non possono appartenere ad Y:

$$R_{\setminus} : \frac{Z = X \setminus Y,\ A = D_Z^- \cup (D_X^- \setminus D_Y^+)}{D'_Z = D_Z \cap [A..D_Z^+]} \tag{7.59}$$

Gli elementi che possono stare in X, possono appartenere a Y oppure a Z. Inoltre, l'insieme Y conterrà tutti gli elementi di X che non possono appartenere a Z.

Come per l'unione e l'intersezione, queste regole vengono applicate solamente nel caso in cui le variabili risultino sufficientemente istanziate.

$$R_{\setminus} : \frac{Z = X \setminus Y,\ Y = \{B\},\ Z = \{C\},\ A = D_X^+ \cap (B \cup C)}{D'_X = D_X \cap [D_X^-..A]} \tag{7.60}$$

$$R_{\setminus} : \frac{Z = X \setminus Y,\ X = \{A\},\ Z = \{C\},\ B = D_Y^- \cup (A \setminus C)}{D'_Y = D_Y \cap [B..D_Y^+]} \tag{7.61}$$

7.5 Consistenza globale e ricerca della soluzione

Facciamo ora alcune considerazioni riguardo la risoluzione globale di un $CSP(\mathcal{FDS})$.

Anzitutto, si noti che il raggiungimento di un **punto fisso** è garantito. Ogni regola di risoluzione incontrata nella sezione precedente infatti:

- *riduce* (eventualmente) i domini delle variabili

- *aggiunge* nuovi vincoli al constraint store solamente quando l'utente lo richiede (*add*)
- *rimuove* (eventualmente) vincoli dallo store
- *sostituisce* (eventualmente) vincoli insiemistici con vincoli di uguaglianza, i quali non aggiungono nuovi vincoli allo store.

Quindi, dal momento in cui l'utente invoca il processo di risoluzione (e quindi non aggiunge ulteriori vincoli allo store), dopo un numero *finito* di iterazioni si avrà fallimento (uno dei domini delle variabili diventa vuoto) oppure non sarà possibile restringere ulteriormente i domini.

In quest'ultimo caso il problema potrebbe essere risolto, fallito o nessuna delle precedenti. Si consideri infatti il seguente esempio:

Esempio 11. Sia $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ un $CSP(\mathcal{FDS})$ dove:

- $\mathcal{V} = \{X,\ Y,\ Z,\ cX,\ cY,\ cZ\}$
- $\mathcal{D}$ tale che $D_X = D_Y = D_Z = [\varnothing..\{0\}]$ e $D_{cX} = D_{cY} = D_{cZ} = [0..1]$
- $\mathcal{C} = \{X \neq Y,\ Y \neq Z,\ Z \neq X\}$.

Si può notare come $\mathcal{P}$ sia non risolto, non fallito ma comunque inconsistente.

L'esempio precedente dimostra come la risoluzione di vincoli su insiemi di interi, basata esclusivamente su regole di riduzione e propagazione dei vincoli, non garantisce la consistenza globale del CSP.
Infatti, $\mathcal{C}$ è già semplificato e le regole $R_{\neq}$ non hanno alcun effetto sul dominio delle variabili di $\mathcal{V}$.
Quindi, il processo di risoluzione terminerà senza effettuare alcuna riduzione e senza individuare l'inconsistenza di $\mathcal{P}$.
In altri termini, il constraint solver non è completo.

Inoltre, anche un CSP consistente è spesso non risolto; quindi, analogamente a quanto descritto nella sezione 4.4, si utilizzeranno apposite tecniche di consistenza globale e ricerca della(e) soluzione(i) basate sul *labeling*.

Fare **labeling** su una n-pla di variabili $V = \langle X_1, \ldots, X_n \rangle \in {\mathcal{V}_{\mathcal{FS}}}^n$ significa assegnare ad ogni variabile insiemistica X_i un valore del proprio dominio.

Come osservato in precedenza, è evidente che attraverso il labeling si possa ottenere la *completezza* della risoluzione; tuttavia, il suo costo computazionale potrebbe risultare decisamente elevato: per questo motivo anche per il labeling su variabili insiemistiche si farà uso di *euristiche* per migliorare l'efficienza della risoluzione.

Le *Value Choice Heuristics* sono di fatto le stesse descritte in 4.4, mentre le *Variable Choice Heuristics* sono solamente riadattate per gestire variabili insiemistiche.

Definizione 7.16 (Variable Choice Heuristic)**.** Sia $k \in \mathbb{N}$ generico.

Una **Variable Choice Heuristic** è una funzione $\rho : {\mathcal{V}_{\mathcal{FS}}}^k \longrightarrow \mathcal{V}_{\mathcal{FS}}$ tale che:

$$\rho(X_1, \ldots, X_k) \in \{X_1, \ldots, X_k\} \quad \text{per ogni } \langle X_1, \ldots, X_k \rangle \in {\mathcal{V}_{\mathcal{FS}}}^k$$

Possibili euristiche ρ per la scelta di una variabile $X \in \{X_1, \ldots, X_n\}$ sono:

left-most:	$\rho(V) = X_1$	
mid-most:	$\rho(V) = X_k,$	dove $k = \left\lfloor \frac{n}{2} \right\rfloor$
right-most:	$\rho(V) = X_n$	
min:	$\rho(V) = X_k,$	dove $D_k^- = min_{\subseteq}\{D_i^- : 1 \leq i \leq n\}$
max:	$\rho(V) = X_k,$	dove $D_k^+ = max_{\subseteq}\{D_i^+ : 1 \leq i \leq n\}$
first-fail:	$\rho(V) = X_k,$	dove $D_k = min_{\#}\{D_i : 1 \leq i \leq n\}$
random:	$\rho(V) = X_k,$	dove $k = random([1..n])$

La differenza fondamentale rispetto al labeling sulle variabili intere non consiste quindi nelle euristiche, bensì nella *strategia di ricerca* della soluzione.

Come già osservato alla fine del capitolo 6 infatti, rimuovere un valore da un set-interval non permette di individuare esattamente ed univocamente altri due set-interval (come invece avviene per gli intervalli di interi).

Si tenga presente inoltre che un set-interval $[A..B]$ contiene esattamente $2^{|B|-|A|}$ elementi, mentre un intervallo di interi $[a..b]$ ne contiene 'solamente' $b - a + 1$. Per questa ragione, istanziare direttamente una variabile insiemistica X scegliendo un valore del suo dominio risulta impraticabile per domini sufficientemente grandi.

La strategia di labeling su una n-pla di variabili $V = \langle X_1, \ldots, X_n \rangle \in {\mathcal{V}_{\mathcal{FS}}}^n$ sarà quindi la seguente:

- seleziono una *variabile* $X = \rho(V)$ con $|D_X| > 1$ e rimuovo X da V
- seleziono un *valore* $x = \lambda(D_X^+ \setminus D_X^-)$[3] che potrebbe appartenere a X
- suddivido il problema corrente $\mathcal{P}$ in due sottoproblemi:

[3]In realtà, questo è un abuso di notazione: anziché $x = \lambda(D_X^+ \setminus D_X^-)$ dovrebbe essere $x = \lambda(y)$ con $D_y = D_X^+ \setminus D_X^-$

(i) $\mathcal{P}'$, ottenuto da $\mathcal{P}$ aggiungendo il vincolo $x * X$

(ii) $\mathcal{P}''$, ottenuto da $\mathcal{P}$ aggiungendo il vincolo $\neg(x * X)$

con $* \in \{\in, \notin\}$

- risolvo ricorsivamente $\mathcal{P}'$ e $\mathcal{P}''$.

Si noti anzitutto che la scelta di $*$ è arbitraria, per cui verrà utilizzata una ulteriore euristica σ tale che:

- se $\sigma =$ first-in, $\mathcal{P}'$ è ottenuto aggiungendo a $\mathcal{P}$ il vincolo $x \in X$, mentre $\mathcal{P}''$ è ottenuto aggiungendo $x \notin X$
- se $\sigma =$ first-nin, $\mathcal{P}'$ è ottenuto aggiungendo a $\mathcal{P}$ il vincolo $x \notin X$, mentre $\mathcal{P}''$ è ottenuto aggiungendo $x \in X$.

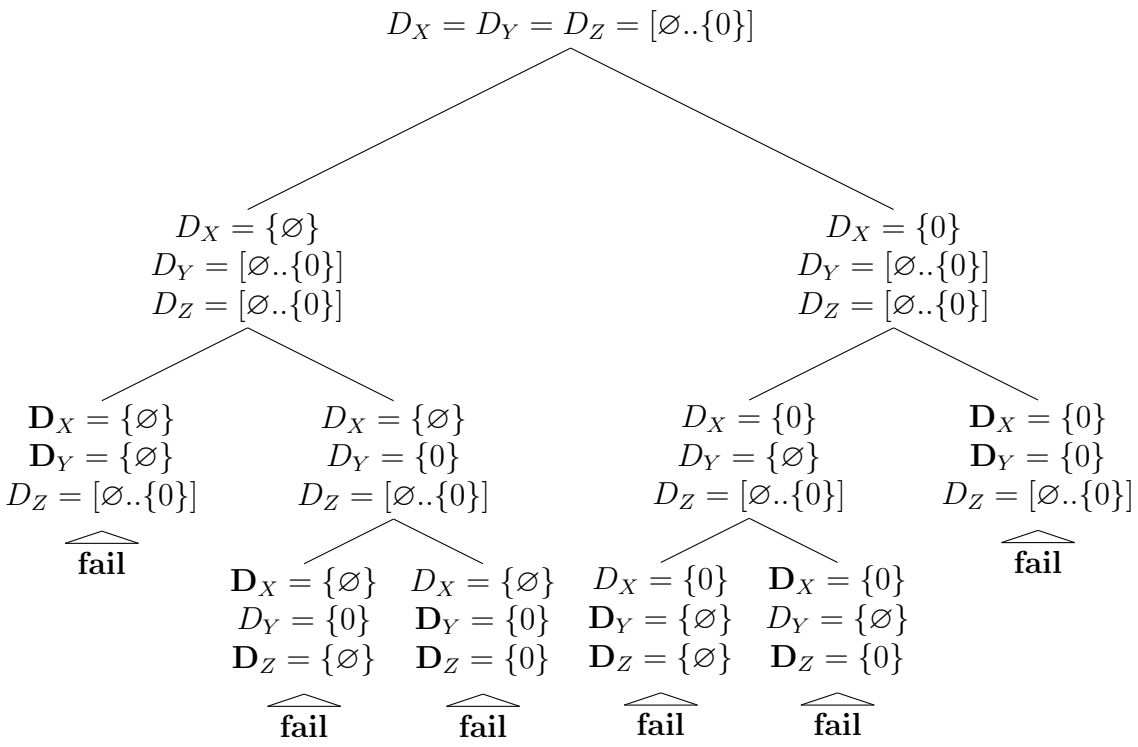
L'approccio è quindi quello di raffinare il dominio di ogni variabile X non *known* mediante disgiunzioni logiche del tipo $x \in X \vee x \notin X$ fino a che X risulti (eventualmente) istanziato. Per meglio comprendere la semantica operazionale di questa strategia illustriamo quindi alcuni esempi.

Esempio 12. Si consideri il CSP dell'esempio 11. In questo caso la *riduzione* del problema (semplificazione del costraint store, applicazione delle regole di riduzione e propagazione dei vincoli) non comporta alcun beneficio.
Vediamo allora come attraverso il labeling sia possibile individuare l'inconsistenza del problema.

Si prenda quindi $V = \langle X, Y, Z\rangle$ con $\rho =$ left-most e $\sigma =$ first-nin.[4]

La risoluzione è descritta dal seguente albero di ricerca, nel quale in ogni nodo vengono indicati i domini di X, Y e Z nel corrispondente sotto-problema.

[4]si noti che in questo caso la scelta di λ è ininfluente, in quanto $|D_v^+ \setminus D_v^-| = 1$ per ogni $v \in \{X, Y, Z\}$



Ogni foglia dell'albero di ricerca corrisponde ad un CSP fallito: non esiste alcuna soluzione, quindi $\mathcal{P}$ è inconsistente.

Si noti inoltre che per individuare l'inconsistenza vengono creati ben *cinque choice-points*, mentre l'analogo problema 1, nel quale le variabili coinvolte sono intere, necessita solamente di un choice-point.

Da ciò si può intuire come il labeling sulle variabili insiemistiche sia decisamente più costoso rispetto a quello sulle variabili intere descritto in 4.4.

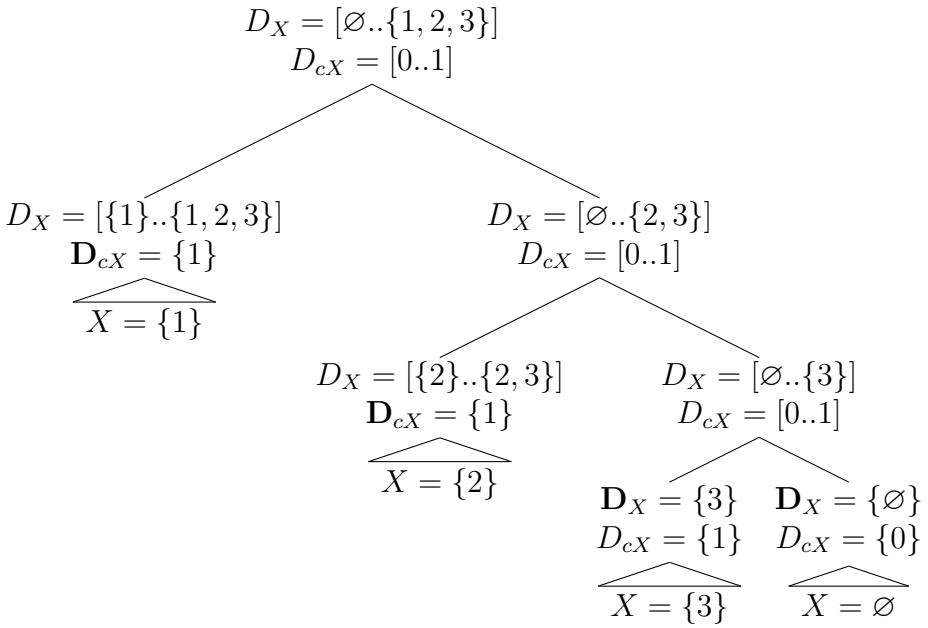
Esempio 13. Si consideri il $CSP(\mathcal{FDS})$ $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$

- $\mathcal{V} = \{X,\ cX\}$
- $\mathcal{D}$ tale che $D_X = [\varnothing..\{1,2,3\}]$ e $D_{cX} = \mathbb{Z}_\alpha$
- $\mathcal{C} = \{\#X :: [0..1]\}$.

Le regole di riduzione dei domini riescono a restringere il dominio di cX nell'intervallo $[0..1]$. Tuttavia, le quattro soluzioni del problema $X = \varnothing$, $X = \{1\}$, $X = \{2\}$ e $X = \{3\}$ non vengono individuate.

Per ottenere tutte le soluzioni di $\mathcal{P}$, è quindi necessario imporre il labeling

su X. Si prenda allora $V = \langle X \rangle$ con $\lambda =$ glb e $\sigma =$ first-in.[5]
L'albero di ricerca risultante è il seguente:



Si può osservare come la cardinalità giochi un ruolo cruciale nella ricerca della soluzione.

Grazie a cZ è possibile infatti ottenere tutte le soluzioni utilizzando solamente *tre choice-points*: non è necessario generare tutti i $2^3 = 8$ possibili valori che può assumere X (ciò comporterebbe l'utilizzo di *sette* choice-points).

[5] si noti che in questo caso la scelta di ρ è ininfluente, essendo coinvolta la sola variabile X

Capitolo 8

JSetL($\mathcal{FDS}$)

In questo capitolo verrà descritta JSetL($\mathcal{FDS}$), un'estensione di JSetL($\mathcal{FD}$) che permette la risoluzione di vincoli su insiemi di interi appartenenti ad un fissato dominio.

8.1 La classe `SetInterval`

La classe `SetInterval` è una struttura dati che permette di rappresentare e manipolare i set-interval di $\mathbb{S}_\beta$.

Per prima cosa conviene notare che, per modellare gli insiemi di interi appartenenti ai set-interval (e quindi contenuti in $\mathbb{Z}_\beta$) si utilizza la classe `MultiInterval` descritta in 5.3.
Ciò non deve stupire: come dimostrato nel teorema 3.2, multi-intervalli e insiemi di interi appartenenti ad un fissato universo sono in corrispondenza biunivoca.
Questa scelta può comportare diversi vantaggi rispetto alla rappresentazione 'classica' degli insiemi come collezione di elementi non ripetuti.

Anzitutto, è possibile memorizzare insiemi *molto grandi* tenendo traccia solamente degli *intervalli* che compongono il corrispondente multi-intervallo. Ad esempio, per rappresentare l'insieme $\mathbb{Z}_\beta$ è sufficiente memorizzare gli estremi $-\beta$ e β anziché tutti i $2\beta + 1$ elementi che appartengono ad esso.

Questa soluzione potrebbe però risultare svantaggiosa nel caso di insiemi particolarmente 'sparsi', ad esempio $A = \{x \in \mathbb{Z}_\beta : x \ mod \ 2 = 0\}$; in questo caso (pessimo), il multi-intervallo corrispondente ad A è dato dall'unione di $|A|$ singoletti disgiunti: non ho alcun risparmio di memoria.

Un altro vantaggio è che il costo delle operazioni insiemistiche tra multi-intervalli non dipende dalla loro cardinalità, bensì dal loro *ordine* (che è sempre minore o uguale alla cardinalità).

Inoltre, tutti gli intervalli contenuti in un multi-intervallo sono *totalmente ordinati* secondo la relazione $\prec$ definita nella sezione 3.2: questa proprietà può risultare utile per ottimizzare le operazioni fra intervalli.

Fatta questa premessa, occupiamoci della classe `SetInterval` vera e propria.

Anzitutto, si osservi che per ogni set-interval $S \in \mathbb{S}_\beta$ si ha che $\varnothing \subseteq S^- $ e $S^+ \subseteq \mathbb{Z}_\beta = [-\beta..\beta]$.

Quindi, analogamente alla classe `Interval`, `SetInterval` conterrà quindi due campi statici ***INF*** e ***SUP*** che rappresentano rispettivamente i valori minimi e massimi (secondo $\subseteq$) che un elemento del set-interval può assumere. Tali campi sono definiti nel seguente modo:

- `public static final MultiInterval` ***INF*** `= new MultiInterval();`
- `public static final MultiInterval` ***SUP*** `=`
  `new MultiInterval(-Interval.SUP / 2, Interval.SUP / 2);`

Ovviamente, ***INF*** è l'insieme vuoto. ***SUP*** invece corrisponde a $\mathbb{Z}_\beta$: ciò significa che il valore della costante β è fissato in `Interval.SUP / 2`.
Questo perché come visto sopra, ogni insieme X che può appartenere ad un set-interval è tale che $\varnothing \subseteq X \subseteq \mathbb{Z}_\beta$ per cui $0 \leq |X| \leq 2\beta + 1$.
Ora, dovendo in seguito risolvere vincoli che coinvolgono la cardinalità di X, tale $|X|$ deve appartenere all'universo $\mathbb{Z}_\alpha = [-\alpha..\alpha]$ definito in 3.2.
Perciò, dev'essere $2\beta + 1 \leq \alpha$ da cui $\beta \leq \dfrac{\alpha - 1}{2}$.
Quindi, come β è stato scelto il $max \left\{ x \in \mathbb{Z} : x \leq \dfrac{\alpha - 1}{2} \right\} = \left\lfloor \dfrac{\alpha - 1}{2} \right\rfloor$.
Essendo α dispari (cfr. sezione 5.2), allora:

$$\beta = \left\lfloor \frac{\alpha - 1}{2} \right\rfloor = \alpha/2 = \texttt{Interval.SUP / 2} = 536870911.$$

Occupiamoci quindi dei principali metodi della classe.

Costruttori

- `SetInterval()`: costruisce un set-interval vuoto $\varnothing$
- `SetInterval(MultiInterval A)`: costruisce il set-interval $\|\{\texttt{A}\}\|_\beta$
- `SetInterval(MultiInterval A, MultiInterval B)`: costruisce il set-interval $[\![\texttt{A..B}]\!]_\beta$

- `SetInterval(Collection<MultiInterval> D)`: costruisce il set-interval corrispondente a $\mathcal{CH}_\beta$(`D`).

Metodi di utilità

- `boolean contains(MultiInterval M)`: ritorna `true` sse `M` $\in$ `this`
- `boolean isEmpty()`: ritorna `true` sse `this` $= \varnothing$
- `boolean isSingleton()`: ritorna `true` sse $|$`this`$| = 1$
- `boolean isUniverse()`: ritorna `true` sse `this` $= [\varnothing..\mathbb{Z}_\beta]$
- `double size()`: ritorna $|$`this`$| = 2^{|\texttt{this}^+|-|\texttt{this}^-|}$ sse

  $$|\texttt{this}^+| - |\texttt{this}^-| \leq \texttt{Double.MAX\_EXPONENT} = 1023$$

  altrimenti ritorna `Double.POSITIVE_INFINITY`
- `MultiInterval getGlb()`: ritorna `this`$^-$ sse `this` $\neq \varnothing$, `null` altrimenti
- `MultiInterval getLub()`: ritorna `this`$^+$ sse `this` $\neq \varnothing$, `null` altrimenti
- `static SetInterval` ***`universe()`***: ritorna $[\varnothing..\mathbb{Z}_\beta]$
- `SetInterval intersect(SetInterval S)`: ritorna il set-interval `this` $\cap$ `S`.

Particolare attenzione va prestata al metodo `size()`. Un set-interval infatti potrebbe contenere un numero esponenziale di elementi, ragion per cui si utilizza il tipo `double` per rappresentarne la dimensione.
Si noti tuttavia che se il set-interval è troppo grande (è il caso per esempio dell'universo $[\varnothing..\mathbb{Z}_\beta]$) viene ritornata la costante `Double.POSITIVE_INFINITY`.

8.2 La classe `FSVar`

La classe `FSVar` permette di modellare le variabili appartenenti a $\mathcal{V}_{\mathcal{FS}}$.
Come `FDVar`, in JSetL questa classe estende la *super-classe* `LVar`.
Vediamone i principali metodi.

Costruttori

Come per le `FDVar`, si tratta di coppie di costruttori: ad ogni variabile è infatti possibile associare un *nome esterno* opzionale.

Di seguito verranno quindi elencati solamente i costruttori senza il parametro di tipo `String` corrispondente al nome esterno.

- `FSVar()`: crea una nuova variabile, con dominio $[\varnothing..\mathbb{Z}_\beta]$
- `FSVar(MultiInterval A)`: crea una nuova variabile, con dominio $\|\{\texttt{A}\}\|_\beta$
- `FSVar(Set<Integer> S)`: crea una nuova variabile, con dominio $\|\{\texttt{S}\}\|_\beta$
- `FSVar(MultiInterval A, MultiInterval B)`: crea una nuova variabile, con dominio $[\![\texttt{A}..\texttt{B}]\!]_\beta$
- `FSVar(Set<Integer> S, Set<Integer> T)`: crea una nuova variabile, con dominio $[\![\texttt{S}..\texttt{T}]\!]_\beta$
- `FSVar(SetInterval S)`: crea una nuova variabile, con dominio `S`
- `FSVar(SetInterval S, Integer k)`: crea una nuova variabile, con dominio `S` e cardinalità `k`
- `FSVar(SetInterval S, MultiInterval M)`: crea una nuova variabile, con dominio `S` e cardinalità avente dominio `M`.

Si noti che come per `FDVar` i costruttori potrebbero sollevare l'eccezione:

`NotValidDomainException`

nel caso si provi a costruire una variabile con dominio vuoto.

Inoltre, alcuni costruttori permettono di definire il dominio delle variabili utilizzando istanze di classi che implementano l'interfaccia `Set<Integer>`, anziché esplicitare i multi-intervalli corrispondenti.

8.2.1 Termini

Come avviene per le `FDVar`, la classe `FSVar` consente di costruire termini composti di $\tau \in \mathcal{T}_{\mathcal{FDS}}$ restituendo vincoli del tipo $x = \tau$.

- `FDVar card()`: ritorna una `FDVar n` tale che $\texttt{n} = \#\texttt{this}$
- `FSVar compl()`: ritorna una `FSVar Z` tale che $\texttt{Z} = \sim\texttt{this}$
- `FSVar intersect(FSVar Y)`: ritorna una `FSVar Z` tale che $\texttt{Z} = X \cap Y$

- `FSVar intersect(MultiInterval A)`: ritorna una `FSVar Z` tale che `Z` $= X \cap A$
- `FSVar union(FSVar) Y`: ritorna una `FSVar Z` tale che `Z` $= X \cup Y$
- `FSVar union(MultiInterval A)`: ritorna una `FSVar Z` tale che `Z` $= X \cup A$
- `FSVar diff(FSVar Y)`: ritorna una `FSVar Z` tale che `Z` $= X \setminus Y$
- `FSVar diff(MultiInterval A)`: ritorna una `FSVar Z` tale che `Z` $= X \setminus A$.

Ad esempio, un termine composto come $|$`X` $\cup \sim$`Y`$|$ con `X` e `Y` `FSVar` si costruisce mediante l'invocazione di:

```
X.union(Y.compl()).card()
```

8.2.2 Vincoli

Vediamo come i seguenti metodi di `FSVar` permettano la costruzione di vincoli atomici appartenenti all'insieme $\mathcal{AC}_{\mathcal{FDS}}$, utilizzando la classe `Constraint` introdotta in 5.4.2.

- `Constraint dom(MultiInterval A, MultiInterval B)`: ritorna il vincolo `this` :: [`A`..`B`]
- `Constraint dom(SetInterval S)`: ritorna il vincolo `this` :: `S`
- `Constraint eq(FSVar Y)`: ritorna il vincolo `this` = `Y`
- `Constraint eq(MultiInterval A)`: ritorna il vincolo `this` = `A`
- `Constraint neq(FSVar Y)`: ritorna il vincolo `this` $\neq$ `Y`
- `Constraint neq(MultiInterval A)`: ritorna il vincolo `this` $\neq$ `A`
- `Constraint disj(FSVar Y)`: ritorna il vincolo `this` $||$ `Y`
- `Constraint disj(MultiInterval A)`: ritorna il vincolo `this` $||$ `A`
- `Constraint strictSubset(FSVar Y)`: ritorna il vincolo `this` $\subset$ `Y`
- `Constraint strictSubset(MultiInterval A)`: ritorna `this` $\subset$ `A`
- `Constraint subset(FSVar Y)`: ritorna il vincolo `this` $\subseteq$ `Y`

- `Constraint subset(MultiInterval A)`: ritorna il vincolo `this` $\subseteq$ `A`.

I vincoli di (non) appartenenza sono invece costruiti invocando i seguenti metodi di `FDVar`:

- `Constraint in(FSVar X)`: ritorna il vincolo `this` $\in$ `X`
- `Constraint in(MultiInterval A)`: ritorna il vincolo `this` :: `A`
- `Constraint nin(FSVar X)`: ritorna il vincolo `this` $\notin$ `X`
- `Constraint nin(MultiInterval A)`: ritorna il vincolo `this.ndom(A)`.

Come si può notare, se il termine insiemistico del vincolo è una costante, ci si può ricondurre ad un vincolo di dominio logicamente equivalente senza costruire un apposito vincolo di (non) appartenenza.

Se per esempio volessi costruire un vincolo del tipo `x` $\in$ `Z` $\setminus$ `Y` con `x` `FDVar` e `Z`, `Y` `FSVar` dovrei scrivere qualcosa come:

```
x.in(Z.diff(Y))
```

8.2.3 Labeling

Come descritto nella sezione 7.5 del capitolo precedente, il labeling sulle variabili insiemistiche sfrutta le euristiche già definite per `FDVar`, utilizzando una ulteriore euristica σ che permette di scegliere se aggiungere prima il vincolo $x \in X$ oppure $x \notin X$.

Se ad esempio volessi impostare ρ = random, λ = lub e σ = first-in, dovrei scrivere qualcosa come:

```
LabelingOptions lop = new LabelingOptions();
lop.var = VarHeuristic.RANDOM;
lop.val = ValHeuristic.LUB;
lop.set = SetHeuristic.FIRST_IN;
```

dove `SetHeuristic` è un *enumerazione* Java definita come:

```
public enum VarHeuristic {
    FIRST_IN,
    FIRST_NIN
}
```

Il valore di default è `SetHeuristic = FIRST_NIN`.

I metodi che `FSVar` offre per il supporto del labeling sono esattamente gli stessi descritti nella sezione 5.4.3 per la classe `FDVar`.

8.2.4 Insiemi parzialmente specificati

La classe `FSVar` permette la definizione di insiemi di interi **parzialmente specificati** analogamente a quanto avviene in CLP($\mathcal{SET}$).

Se $x_1, \ldots, x_n \in \mathcal{V}_{\mathcal{FD}}$ con $n \geq 0$ e $X, Z \in \mathcal{V}_{\mathcal{FS}}$ allora è possibile risolvere vincoli del tipo:

$$Z = \{x_1, \ldots, x_n \mid X\} \stackrel{def}{=} \{x_1\} \cup \ldots \cup \{x_n\} \cup X$$

Per costruire vincoli di questo tipo, si utilizzano i seguenti metodi:

- `eq(FDVar x)`: costruisce il vincolo `this` $= \{$`x`$\}$
- `eq(Collection<FDVar> v)`: costruisce il vincolo `this` $= \{x_1, \ldots, x_n\}$ se `v` è una `Collection` di variabili intere $[x_1, \ldots, x_n]$
- `eq(FDVar[] v)`: come sopra, con `v` *array*
- `eq(FDVar x, FSVar X)`: costruisce il vincolo `this` $= \{$`x` $\mid$ `X`$\}$
- `eq(Collection<FDVar> v, FSVar X)`: costruisce il vincolo `this` $= \{x_1, \ldots, x_n \mid$ `X`$\}$ se `v` è una `Collection` di variabili intere $[x_1, \ldots, x_n]$
- `eq(FDVar[] v, FSVar X)`: come sopra, con `v` *array*.

Nell'implementazione corrente, un vincolo del tipo $Z = \{x_1, \ldots, x_n \mid X\}$ viene semplicemente espanso in n unioni:

$$Z = X_1 \ \cup \ \ldots \ \cup \ X_n \ \cup \ X \qquad \text{dove } X_i = \{x_i\} \text{ per } i = 1, \ldots, n$$

Si noti che ognuna di queste unioni comporta l'inserimento nel constraint store di ulteriori vincoli (cfr. regola $R_{\cup}$ 7.45).
Inoltre, per poter rappresentare i singoletti $X_i = \{x_i\}$ è necessario aggiungere allo store tre vincoli atomici per ogni $i = 1, \ldots, n$:

$$x_i \in X_i \ \wedge \ \#X_i = 1 \ \wedge \ x_i \in Z$$

Ciò comporta un notevole carico di vincoli sullo store: un possibile sviluppo futuro potrebbe permettere una trattazione più completa ed efficiente di questo particolare tipo di insiemi.

Tuttavia, si noti che nel caso in cui il dominio delle variabili coinvolte non risulti particolarmente ampio, attraverso il labeling su di esse è possibile ottenere tempi di risoluzione decisamente migliori rispetto a CLP($\mathcal{SET}$).

Un esempio di utilizzo di insiemi parzialmente specificati è riportato nella sezione 8.3.1.

Altri metodi di utilità

Vediamo infine alcuni metodi ausiliari della classe `FSVar`.

- `boolean equals(FDVar x)`: ritorna `true` sse `this` = `x`
- `SetInterval getDomain()`: ritorna $D_{\texttt{this}}$
- `void output()`: stampa `this` su *standard output*, incluse informazioni sul suo dominio se $|D_{\texttt{this}}| > 1$.

8.3 Esempi

Concludiamo il capitolo illustrando qualche esempio completo di programmi che utilizzano le funzionalità di JSetL($\mathcal{FDS}$).

8.3.1 Permutazioni

Consideriamo ancora una volta l'esempio delle permutazioni, fornendo un'implementazione che risolve il problema mediante l'utilizzo di un insieme *parzialmente specificato* di elementi.

```
import JSetL.*;

class PermutationsFDS {

    public static void main (String[] args)
    throws Failure {
        // Constraint solver.
        SolverClass solver = new SolverClass();

        // x, y, z::[1..3].
        FDVar x = new FDVar("x", 1, 3);
        FDVar y = new FDVar("y", 1, 3);
        FDVar z = new FDVar("z", 1, 3);
        FDVar[] vars = {x, y, z};

        // X = [1..3].
        FSVar X = new FSVar("X", new MultiInterval(1, 3));

        // {x, y, z} = [1..3].
        solver.add(X.eq(vars));
```

```
        // Labeling on {x, y, z}.
        solver.add(FDVar.label(vars));

        // Try to find a solution.
        solver.solve();

        // Print all solutions.
        int i = 0;
        do {
            ++i;
            System.out.println("Solution no. " + i);
            x.output();
            y.output();
            z.output();
        } while (solver.nextSolution());
    }
}
```

Come si può notare, rispetto all'esempio 5.6.1 non si utilizza un vincolo `allDifferent` che vincola x, y, z ad essere tutte diverse fra loro.

In questo caso, l'insieme parzialmente specificato $\{x, y, z\}$ viene uguagliato a $[1..3]$.

Il risultato, ovviamente, è il medesimo: per soddisfare tale uguaglianza le variabili x, y e z devono essere tutte distinte.
La parte riguardante la stampa delle soluzioni è quindi omessa.

8.3.2 Triple di Steiner

Il problema delle triple di Steiner di ordine n consiste nel trovare un insieme di $t = \frac{n(n-1)}{6}$ triple tali che:

- ogni tripla contiene esattamente tre elementi distinti
- ogni elemento appartiene all'insieme $\{1, \ldots, n\}$
- ogni coppia di triple distinte ha al più un elemento in comune.

Si tratta di un problema di matematica combinatoria, nel quale n deve essere tale che $(n \ mod \ 6) \in \{1, 3\}$.

Formalmente, si tratta si trovare t insiemi S_i che soddisfino le seguenti condizioni:

$$\begin{array}{ll} S_i \subseteq \{1, \ldots, n\} \text{ e } |S_i| = 3 & \text{per } i = 1, \ldots, t \\ |S_i \cap S_j| \leq 1 & \text{per } 1 \leq i < j \leq t \end{array}$$

Il codice JSetL che implementa tali specifiche, con $n = 7$, è il seguente:

```
import JSetL.*;

public class TernarySteiner {

    // Ternary Steiner order (N modulo 6 has to be 1 or 3).
    private static final int N = 7;
    // Number of Steiner triples.
    private static final int T = (N * (N - 1)) / 6;

    public static void main(String[] args)
    throws Failure {
        FSVar[] s = new FSVar[T];
        SolverClass solver = new SolverClass();
        MultiInterval lb = new MultiInterval();
        MultiInterval ub = new MultiInterval(1, N);

        // S_i::[{}..[1..N]] AND #S_i = 3 for i = 0, ..., T - 1.
        for (int i = 0; i < T; ++i) {
            FSVar var = new FSVar("S_" + i, lb, ub);
            solver.add(var.card().eq(3));
            solver.add(var.label());
            s[i] = var;
        }

        // #(S_i inters S_j) <= 1 for 0 <= i < j < T.
        for (int i = 0; i < T - 1; ++i)
            for (int j = i + 1; j < T; ++j)
                solver.add(s[i].intersect(s[j]).card().le(1));

        // Try to find a solution.
        solver.solve();

        // Print one solution.
        System.out.println( "***** N = " + N + " ***** \n");
```

```
        for (int i = 0; i < T; ++i)
            s[i].output();
    }
}
```

L'output prodotto è il seguente:

```
***** N = 7 *****

S_0 = [5..7]
S_1 = {3..4, 7}
S_2 = {2, 4, 6}
S_3 = {2..3, 5}
S_4 = {1, 4..5}
S_5 = {1, 3, 6}
S_6 = {1..2, 7}
```

Si può osservare come la formalizzazione dei vincoli del problema non sia sufficiente per raggiungere una soluzione; come già osservato infatti il solver non è completo, per cui è necessario l'utilizzo del labeling sulle variabili insiemistiche S_i.

Si noti che per ottenere una risoluzione efficiente all'aumentare di n è essenziale utilizzare euristiche di labeling mirate.
Infatti, in questo problema sono presenti molte *simmetrie*: ogni coppia di triple distinte S_i ed S_j può essere scambiata ottenendo di fatto una soluzione equivalente.
Quindi, per trovare la strategia migliore è utile studiare le proprietà matematiche del problema (ad esempio, è dimostrato che ogni elemento appartiene esattamente a $\frac{n-1}{2}$ triple).

Nell'esempio tuttavia si è deciso di mantenere le impostazioni di default per il labeling, non essendo interessati allo studio di questo particolare problema.

In generale, come si vedrà anche in seguito, trovare una strategia di labeling che minimizzi le simmetrie è *cruciale* per ottenere una risoluzione efficiente: in questo modo, non si perde tempo per esplorare cammini 'inutili' nell'albero di ricerca.

8.3.3 Weighted Hamming Codes

Il problema dei *Weighted Hamming Codes* può essere così formulato: dati quattro parametri interi n, b, d e w trovare n stringhe binarie tali che:

- ogni stringa ha esattamente b bits
- la *distanza di Hamming*[1] $d(S_i, S_j)$ tra ogni coppia di stringhe distinte S_i ed S_j è tale che $d(S_i, S_j) \geq d$
- ogni stringa pesa w (cioè, il numero di bit a '1' della stringa è esattamente w).

Una possibile formalizzazione del problema consiste nel modellare le stringhe binarie attraverso n variabili insiemistiche $S_1, \ldots, S_n$ tali che:

$$\begin{array}{ll} S_i :: [\varnothing..[1..b]] \text{ e } |S_i| = w & \text{per } i = 1, \ldots, n \\ |S_i \cap S_j| + |[1..b] \setminus (S_i \cup S_j)| \leq b - d & \text{per } 1 \leq i < j \leq n \end{array}$$

Vediamo quindi un esempio di implementazione con $n = 6$, $b = 10$, $d = 6$ e $w = 5$:

```
import JSetL.*;

public class WeightedHammingCodes {

    // Number of distinct codewords.
    private static final int N = 6;

    // Length of each codeword.
    private static final int B = 10;

    // Minimum Hamming distance between distinct codewords.
    private static final int D = 6;

    // Weight of each codeword.
    private static final int W = 5;

    public static void main(String[] args)
    throws Failure {
        MultiInterval lb = new MultiInterval();
        MultiInterval ub = new MultiInterval(1, B);
        FSVar[] s = new FSVar[N];
        SolverClass solver = new SolverClass();
```

[1]Ricordiamo che, date due stringhe binarie di l bit u e v, la loro distanza di Hamming $d(u, v)$ è il numero di posizioni nelle quali i simboli corrispondenti sono diversi. Ovviamente, $0 \leq d(u, v) \leq l$

```
        // S_i :: [{}..[1..B]] AND #S_i = W
        for (int i = 0; i < N; ++i) {
            FSVar tmp = new FSVar(lb, ub);
            s[i] = tmp;
            solver.add(tmp.label());
            solver.add(tmp.card().eq(W));
        }

        // #(S_i inters S_j) + #([1..B] \ (S_i union S_j))
        // <= B - D.
        for (int i = 0; i < N - 1; ++i)
            for (int j = i + 1; j < N; ++j)
                solver.add(s[i].intersect(s[j]).card()
                .sum(new FSVar(ub).diff(s[i].union(s[j])).card())
                .le(B - D));

        // Try to find a solution.
        solver.solve();

        // Print one solution.
        System.out.print("S = {");
        for (int i = 0; i < N - 1; ++i)
            System.out.print(toCodeword(s[i].getValue()) + ", ");
        System.out.print(toCodeword(s[N - 1].getValue()) + "}");
    }
}
```

Nel codice è stata omessa la funzione `toCodeword` che permette la stampa delle stringhe risultanti in un formato leggibile dall'utente.
Nel nostro esempio, l'output sarà:

```
S = {0000011111, 0011100011, 0101101100,
     1010110100, 1101010001, 1110001010}
```

Anche in questo caso, per ottenere (eventualmente) una soluzione è necessario forzare il labeling sulle variabili insiemistiche.

Inoltre, la presenza di simmetrie implica l'utilizzo di apposite strategie che permettano la risoluzione in tempi ragionevoli all'aumentare della dimensione del problema.

Questo tipo di problematiche verranno trattate più approfonditamente in seguito.

Capitolo 9

Esempi e test

In questo capitolo verranno affrontati due CSP ben noti in letteratura, il problema delle *n-regine* e quello dei *social golfers*, allo scopo di valutarne le prestazioni al variare della dimensione del problema.

Dopo aver introdotto tali problemi, verrà quindi presentato il codice JSetL che li risolve e verranno effettuate considerazioni riguardo i tempi di risoluzione.

La macchina sulla quale verranno eseguiti i test utilizza il sistema operativo Fedora 10, processore Intel Pentium Duo a 1.86 GHz e 2 GB di RAM.

9.1 Il problema delle n regine

Il problema delle n regine è un classico esempio di CSP contenente vincoli su interi.

Fissato un intero $n > 3$, tale problema consiste nel posizionare n regine su una scacchiera $n \times n$ in modo che nessuna di esse possa attaccarne un'altra.

In altri termini, nessuna regina può trovarsi sulla stessa riga, colonna o diagonale di un'altra.

Il problema si può quindi modellare con un CSP $\mathcal{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ dove:

- $\mathcal{V} = \{x_1, \ldots, x_n\}$ tale che $x_i = j$ sse sulla scacchiera è presente una regina nella posizione di coordinate (i, j) (in questo modo impongo implicitamente che sulla stessa riga non possano stare due regine, essendo che ogni x_i non può assumere due valori distinti).

- $\mathcal{D} = D_1 \times \ldots \times D_n$ tale che $D_i = [1..n]$ per $i = 1, \ldots, n$

- $\mathcal{C}$ è costituito dai seguenti vincoli per $1 \leq i < j \leq n$:

$$\begin{aligned} x_i &\neq x_j && \text{(nessuna regina sulla stessa colonna)} \\ x_i - x_j &\neq i - j && \text{(nessuna regina sulla stessa diagonale } \searrow) \\ x_i - x_j &\neq j - i && \text{(nessuna regina sulla stessa diagonale } \nearrow) \end{aligned}$$

Si noti che i vincoli di $\mathcal{C}$ sono di fatto logicamente equivalenti ai vincoli:

$$\begin{aligned} &\mathit{allDifferent}(x_1, \ldots, x_n) \\ &\mathit{allDifferent}(x_1 - 1, \ldots, x_n - n) \\ &\mathit{allDifferent}(x_1 + 1, \ldots, x_n + n) \end{aligned}$$

dove in generale:

$$\mathit{allDifferent}(v_1, \ldots, v_n) = \bigwedge_{1 \leq i < j \leq n} v_i \neq v_j$$

Attraverso JSetL è possibile descrivere e risolvere questo problema al variare di n.

Il seguente codice fornisce un'implementazione del problema utilizzando un'indicizzazione del tipo $0, \ldots, n-1$ anziché $1, \ldots, n$: ciò permette una gestione più naturale degli array coinvolti, senza ledere la generalità del problema.

```
import java.io.*;
import JSetL.*;

class Queens {
   public static final int N = 10;
   public static SolverClass solver = new SolverClass();

   public static void main (String[] args)
   throws IOException,Failure {
       int i, j;

       // x[i] = j iff there is a queen in the position (i, j).
       FDVar[] x = new FDVar[N];
       FDVar[] y = new FDVar[N];
       FDVar[] z = new FDVar[N];

       // y[0] = x[0] - 0 = x[0] = x[0] + 0 = z[0]
```

```
FDVar tmp = new FDVar(0, N - 1);
x[0] = tmp;
y[0] = tmp;
z[0] = tmp;
for (i = 1; i < N; ++i) {
    // Associates to each variable the corresponding domain.
    x[i] = new FDVar(0, N - 1);
    y[i] = new FDVar(-i, N - 1 - i);
    z[i] = new FDVar(i, N - 1 + i);
    // y[i] = x[i] - i <===> y[i] - x[i] = -i
    solver.add(y[i].sub(x[i]).eq(-i));
    // z[i] = x[i] + i <===> z[i] - x[i] =  i
    solver.add(z[i].sub(x[i]).eq(i));
    // Try to assign to x[i] the median value of its domain.
    solver.add(x[i].label(ValHeuristic.MEDIAN));
}

// allDifferent constraints on x, y and z.
solver.add(FDVar.allDifferent(x));
solver.add(FDVar.allDifferent(y));
solver.add(FDVar.allDifferent(z));

// Try to find a solution.
solver.solve();

// Print the coordinates of each queen on the board.
for (i = 0; i < N; ++i) {
    System.out.print("\n Queen_" + i + " in ");
    System.out.print("(" + i + "," + x[i].toString() + ")");
}

// Print the board.
System.out.print("\n \n");
for (i = 0; i < N; ++i) {
    int queen = x[i].getValue();
    j = 0;
    for (; j < queen; ++j)
        System.out.print(" . ");
    System.out.print(" @ ");
    for (j = queen + 1; j < N; ++j)
        System.out.print(" . ");
```

```
            System.out.println();
        }
        return;
    }
}
```

L'output prodotto, con $n = 10$, è il seguente:

```
Queen_0 in (0,8)
Queen_1 in (1,4)
Queen_2 in (2,7)
Queen_3 in (3,3)
Queen_4 in (4,0)
Queen_5 in (5,9)
Queen_6 in (6,1)
Queen_7 in (7,5)
Queen_8 in (8,2)
Queen_9 in (9,6)

 .  .  .  .  .  .  .  .  @  .
 .  .  .  .  @  .  .  .  .  .
 .  .  .  .  .  .  .  @  .  .
 .  .  .  @  .  .  .  .  .  .
 @  .  .  .  .  .  .  .  .  .
 .  .  .  .  .  .  .  .  .  @
 .  @  .  .  .  .  .  .  .  .
 .  .  .  .  .  @  .  .  .  .
 .  .  @  .  .  .  .  .  .  .
 .  .  .  .  .  .  @  .  .  .
```

Facciamo alcune considerazioni. Per $i = 0, \ldots, n-1$ si ha che:

- l'array `x` contiene le variabili del problema: `x[`i`]` $= x_{i+1}$
- l'array `y` è tale che `y[`i`]` $-$ `x[`i`]` $= -i$, cioè `y[`i`]` $=$ `x[`i`]` $- i$
- l'array `z` è tale che `z[`i`]` $-$ `x[`i`]` $= i$, cioè `z[`i`]` $=$ `x[`i`]` $- i$
- attraverso il metodo `allDifferent` impongo i vincoli di diverso su `x`, `y` e `z`.

Si è verificato sperimentalmente che una tale definizione (logicamente equivalente) dei vincoli permette di ottenere migliori performances nella risoluzione.

Tuttavia, ancora una volta si può osservare come l'efficienza al crescere di n sia fortemente influenzata dalla strategia di *labeling* adottata (anche il problema delle n-regine è infatti ricco di simmetrie).

Come si può notare in Figura 9.1, utilizzando le euristiche *glb* o *lub* il costo (pressoché equivalente) aumenta in modo considerevole al crescere di n. L'euristica *mid-most* è sicuramente migliore di queste; tuttavia, dal grafico risulta evidente come *median* sia l'euristica che determina i risultati migliori.

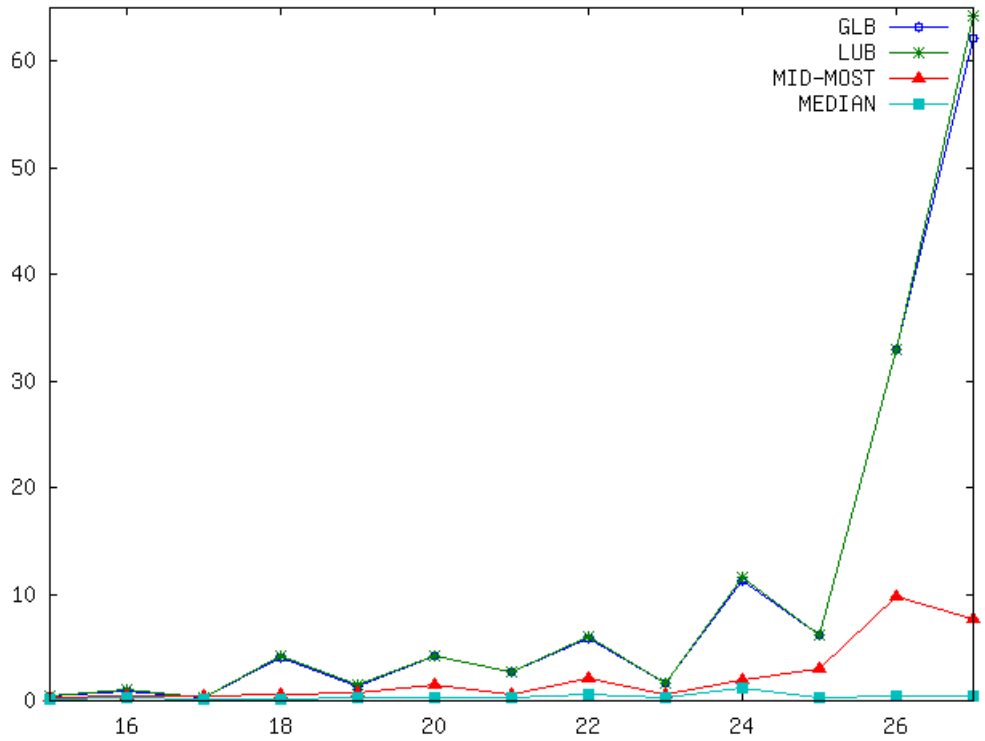


Figura 9.1: Tempo impiegato (in secondi) per raggiungere la prima soluzione, utilizzando diverse euristiche di labeling al variare di n tra 15 e 27

L'utilizzo dell'euristica *median* ha contribuito ad ottenere risultati soddisfacenti al crescere di n.

Si noti infatti che la precedente implementazione di JSetL($\mathcal{FD}$) descritta in [14] già a partire da $n = 18$ presentava tempi di risoluzione non ragionevoli (superiori ai tre minuti).

Ora, grazie all'introduzione di multi-intervalli ed euristiche di labeling (oltre ad altre ottimizzazioni interne alla libreria) la situazione è decisamen-

te migliorata (cfr. Tabella 9.1, Figura 9.2).

Come si può notare dal grafico 9.2, la curva dei tempi di calcolo non segue un andamento *monotono*.
Infatti, nonostante una certa proporzionalità tra n ed il costo di risoluzione $T(n)$, non vale in generale che $n_1 < n_2 \Longrightarrow T(n_1) \leq T(n_2)$.
Ciò è ben visibile osservando il grafico: la curva $T(n)$ presenta numerosi 'picchi'; in particolare, per alcuni valori di n particolarmente 'sfortunati' la risoluzione non converge in tempi ragionevoli.

Nella Tabella 9.1 le righe corrispondenti a tali valori sono marcate in grassetto, mentre nel grafico si possono notare 'buchi' nella funzione (per tutti gli n tali che $T(n) > 3$ minuti, si è posto $T(n) = +\infty$).
Ovviamente, al crescere di n aumenta anche la frequenza di queste configurazioni (si osservi ad esempio quanto accade per $130 \leq n \leq 140$).

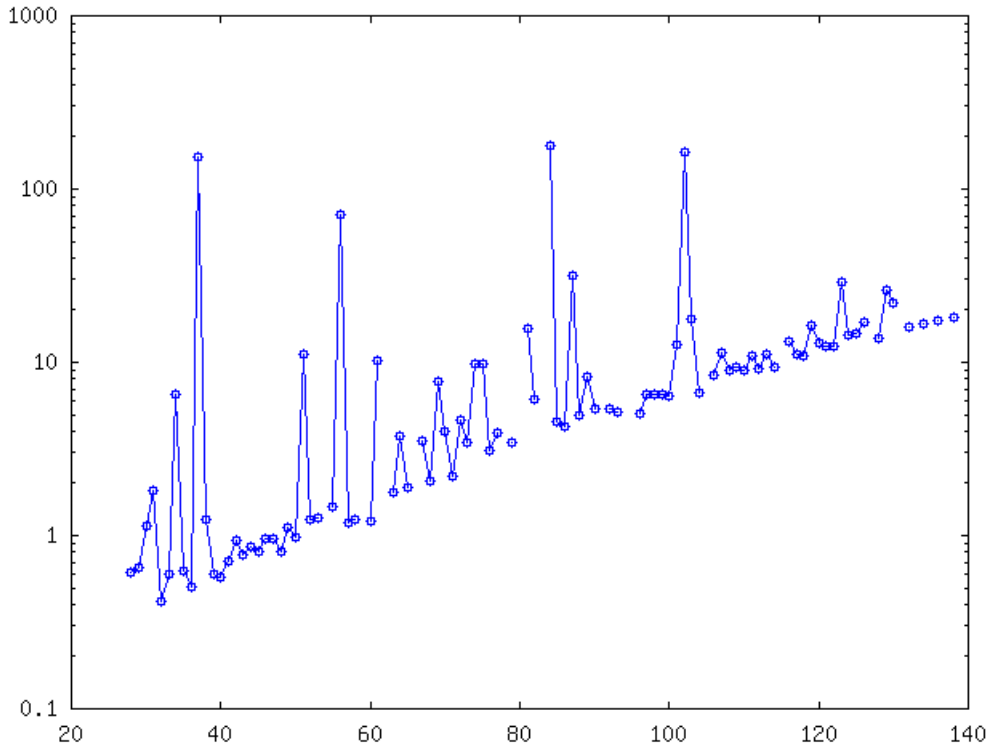


Figura 9.2: Tempo impiegato per raggiungere la prima soluzione, utilizzando l'euristica median al variare di n tra 28 e 140. Si noti che, per motivi di visualizzazione grafica, si è utilizzata una scala logaritmica per rappresentare l'asse delle ordinate.

n	$T(n)$	n	$T(n)$	n	$T(n)$	n	$T(n)$	n	$T(n)$
28	0.613	51	11.173	74	9.768	97	6.468	119	16.224
29	0.655	52	1.232	75	9.79	98	6.553	120	12.786
30	1.139	53	1.271	76	3.095	99	6.54	121	12.454
31	1.818	**54**	-	77	3.896	100	6.379	122	12.313
32	0.417	55	1.474	**78**	-	101	12.776	123	29.084
33	0.595	56	70.579	79	3.442	102	161.968	124	14.295
34	6.59	57	1.185	**80**	-	103	17.834	125	14.753
35	0.625	58	1.231	81	15.55	104	6.649	126	17.065
36	0.507	**59**	-	82	6.124	**105**	-	**127**	-
37	154.456	60	1.217	**83**	-	106	8.357	128	13.736
38	1.229	61	10.279	84	177.414	107	11.371	129	25.982
39	0.605	**62**	-	85	4.557	108	9.085	130	21.81
40	0.574	63	1.772	86	4.249	109	9.327	**131**	-
41	0.716	64	3.755	87	31.838	110	9.084	132	16.092
42	0.944	65	1.904	88	4.916	111	10.819	**133**	-
43	0.771	**66**	-	89	8.257	112	9.196	134	16.702
44	0.856	67	3.522	90	5.406	113	11.211	**135**	-
45	0.802	68	2.066	**91**	-	114	9.282	136	17.479
46	0.957	69	7.778	92	5.391	**115**	-	**137**	-
47	0.953	70	4.008	93	5.161	116	13.28	138	18.282
48	0.816	71	2.212	**94**	-	117	11.221	**139**	-
49	1.121	72	4.651	**95**	-	118	10.79	**140**	-
50	0.983	73	3.424	96	5.077				

Tabella 9.1: Tempo impiegato (in secondi) per raggiungere la prima soluzione, utilizzando l'euristica median al variare di n tra 28 e 140

É importante sottolineare che i vincoli di tipo *allDifferent*$(x_1, \ldots, x_n)$ utilizzati in questi test vengono risolti semplicemente generando $\frac{n(n-1)}{2}$ vincoli di diverso $x_i \neq x_j$ per $1 \leq i < j \leq n$.

Ciò comporta un carico notevole sul constraint store ed una perdita di informazione relazionale sulle n variabili coinvolte.

Questo approccio è sicuramente migliorabile attraverso una trattazione *globale* del vincolo *allDifferent*.
Ad esempio, si potrebbe estendere quanto descritto in [14] per insiemi di variabili intere aventi come dominio un multi-intervallo, senza utilizzare strutture dati ausiliarie come `LSet`.

9.2 Il problema dei Social Golfers

Occupiamoci ora di un CSP contenente vincoli insiemistici.

Dati tre parametri interi g, s e w il problema dei Social Golfers consiste nel suddividere un insieme di $g \times s$ golfisti in g gruppi composti da s giocatori ciascuno per un totale di w settimane, in modo che ogni coppia di giocatori capiti nello stesso gruppo al più una volta.

Formalmente, questo problema consiste nel trovare una matrice di $w \times g$ insiemi di interi S_{ij} tali che:

$$\begin{aligned} S_{ij} :: [\varnothing..[1..g \cdot s]] \text{ e } |S_{ij}| = s \qquad & \text{per} \quad i = 1, \ldots, w \text{ e } j = 1, \ldots, g \\ \bigcup_{k=1,\ldots,g} S_{ik} = [1..g \cdot s] \qquad & \text{per} \quad i = 1, \ldots, w \\ |S_{ih} \cap S_{jk}| \leq 1 \qquad & \text{per} \quad 1 \leq i < j \leq w \text{ e } h, k = 1, \ldots, g \end{aligned}$$

Di fatto, per ogni settimana l'insieme corrispondente a tutti i golfisti $[1..g \cdot s]$ dev'essere *partizionato* in g sottoinsiemi di esattamente s elementi ciascuno. Inoltre, prese due settimane distinte w_i e w_j, ogni partizione di w_i può avere *al più* un elemento in comune con ogni altra partizione di w_j.

Vediamo una possibile implementazione del problema, ricordando che, come per le n-regine, gli indici partono da 0 anziché da 1.

```
import java.util.Vector;
import JSetL.*;

public class SocialGolfers {

    // Number of weeks.
    private static final int W = 5;
    // Number of groups.
    private static final int G = 4;
    // Number of golfers for each group.
    private static final int S = 4;

    public static void main(String[] args)
    throws Failure {
        MultiInterval lb = new MultiInterval();
        MultiInterval ub = new MultiInterval(1, G * S);
        FSVar[][] groups = new FSVar[W][G];
        SolverClass solver = new SolverClass();
        int i, j, h, k;
```

```
// #(groups[i][j]) = S AND for each i = 0, ..., W - 1
// union_j(groups[i][j]) = {1, ..., G * S}
for (i = 0; i < W; ++i) {
    Vector<FDVar> week = new Vector<FDVar>(G * S);
    for (j = 0; j < G; ++j) {
        FSVar var = new FSVar(lb, ub);
        solver.add(var.card().eq(S));
        groups[i][j] = var;
        for (k = 0; k < S; ++k) {
            FDVar tmp = new FDVar(1, G * S);
            solver.add(tmp.in(var));
            solver.add(tmp.label());
            week.add(tmp);
        }
    }
    solver.add(FDVar.allDifferent(week));
 }

// for each i = 0, ..., W - 2     AND
// for each j = i + 1, ..., W - 1 AND
// for each h, k = 0, ..., G - 1
// #(groups[i][h] inters groups[j][k]) <= 1.
for (i = 0; i < W - 1; ++i)
    for (j = i + 1; j < W; ++j)
        for (h = 0; h < G; ++h)
            for (k = 0; k < G; ++k)
                solver.add(groups[i][h]
                .intersect(groups[j][k])
                           .card().le(1));

// Try to find a solution.
solver.solve();

System.out.println( "***** W = " + W +
                    " ***** G = " + G +
                    " ***** S = " + S + " ***** \n");

for (i = 0; i < W; ++i) {
    System.out.println("Week no. " + (i + 1) + ":");
    for (j = 0; j < G; ++j)
```

```
                System.out.println("\tGroup no."
                                    + (j + 1) + ": " + groups[i][j]);
        }
    }
}
```

Presi ad esempio $w = 5$, $g = 4$ e $s = 4$, l'output prodotto è il seguente:

```
***** W = 5 ***** G = 4 ***** S = 4 *****

Week no. 1:
    Group no.1: [1..4]
    Group no.2: [5..8]
    Group no.3: [9..12]
    Group no.4: [13..16]
Week no. 2:
    Group no.1: {1, 5, 9, 13}
    Group no.2: {2, 6, 10, 14}
    Group no.3: {3, 7, 11, 15}
    Group no.4: {4, 8, 12, 16}
Week no. 3:
    Group no.1: {1, 6, 11, 16}
    Group no.2: {2, 5, 12, 15}
    Group no.3: {3, 8..9, 14}
    Group no.4: {4, 7, 10, 13}
Week no. 4:
    Group no.1: {1, 7, 12, 14}
    Group no.2: {2, 8, 11, 13}
    Group no.3: {3, 5, 10, 16}
    Group no.4: {4, 6, 9, 15}
Week no. 5:
    Group no.1: {1, 8, 10, 15}
    Group no.2: {2, 7, 9, 16}
    Group no.3: {3, 6, 12..13}
    Group no.4: {4..5, 11, 14}
```

Per prima cosa, conviene notare che anziché codificare il vincolo di unione globale:

$$\bigcup_{k=1,\ldots,g} S_{ik} = [1..g \cdot s]$$

ogni insieme S_{ij} viene definito attraverso s variabili intere $x_{ij}^k :: [1..g \cdot s]$ nel

seguente modo:

$$S_{ij} = \{x_{ij}^1, \ldots, x_{ij}^s\}$$

Quindi, per $i = 1, \ldots, w$ si pone *allDifferent*$(x_{i1}^1, \ldots, x_{i1}^s, \ldots, x_{ig}^1, \ldots, x_{ig}^s)$. In questo modo, si ha che:

$$\bigcup_{k=1,\ldots,g} S_{ik} = \{x_{i1}^1, \ldots, x_{i1}^s, \ldots, x_{ig}^1, \ldots, x_{ig}^s\} = [1..g \cdot s]$$

Una tale scelta è dovuta al fatto che l'attuale implementazione non gestisce il vincolo 'generalizzato' di unione: $\bigcup : \mathcal{P}^2(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ per cui un vincolo del tipo

$$X = \bigcup\{X_1, \ldots, X_n\} \qquad \text{con } X, X_1, \ldots, X_n \subseteq \mathbb{Z}$$

andrebbe espanso in $n - 1$ vincoli di unione: $X = X_1 \cup \ldots \cup X_n$.

In questo particolare problema, tale limitazione viene 'aggirata' riducendosi alla risoluzione di vincoli su interi, tipicamente meno costosa rispetto a quella su insiemi di interi.

La soluzione è quindi ottenuta facendo labeling sulle variabili x_{ij}^k anziché sugli insiemi S_{ij}.

Ancora una volta, diventa cruciale adottare un'adeguata strategia di ricerca per ottenere tempi di calcolo ragionevoli all'aumentare della dimensione del problema.

Con le sole euristiche *non-randomizzate* (glb, lub, median, mid-most) si ottengono infatti tempi di risoluzione molto distanti da quelli riportati ad esempio in [3].

Si è quindi deciso di testare l'euristica *mid-random*, che permette di assegnare ad una variabile x il valore medio di un intervallo di D_x scelto in modo casuale ed equiprobabile.

Così facendo, si è potuto verificare sperimentalmente come effettivamente le euristiche di labeling influenzino in modo determinante l'efficienza della risoluzione.

Una caratteristica interessante dell'euristica mid-random applicata a questo problema è che, nel caso la risoluzione termini in tempi ragionevoli (nell'ordine di pochi secondi) i risultati ottenuti sono spesso migliori di quelli citati nell'articolo [3].
Di contro, l'utilizzo di una strategia randomizzata potrebbe comportare tempi di attesa *indefiniti* (si può osservare empiricamente come all'aumentare della dimensione del problema diminuisca la probabilità di trovare una soluzione in tempi ragionevoli).

Ovviamente, i risultati riportati (cfr. Tabella 9.2) non implicano che JSetL($\mathcal{FDS}$) offra performances migliori rispetto ai risolutori citati (anche

perché i test sono effettuati su macchine differenti).
Tuttavia essi forniscono una dimostrazione pratica della fondamentale importanza, per problemi di una certa complessità, dell'utilizzo di euristiche che minimizzino lo spazio di ricerca.

Inoltre, potrebbe risultare interessante lo studio di *strategie randomizzate* come alternativa alle strategie 'deterministiche' classiche.

$w-g-s$	JSetL($\mathcal{FDS}$)	Cardinal	ic_sets	ROBDD
$2-5-4$	0.329	0.83	5.3	0.1
$2-6-4$	0.464	1.75	35.5	0.2
$2-7-4$	0.524	2.82	70.3	0.6
$2-8-5$	0.741	-	-	3.1
$3-5-4$	0.646	1.89	9.3	0.5
$3-6-4$	0.921	4.62	59.2	2.3
$3-7-4$	0.812	6.37	113.6	3.5
$4-5-4$	-	3.13	10.5	1.3
$4-6-5$	-	-	-	171.5
$4-7-4$	1.375	12.46	135.8	21.8
$4-9-4$	3.341	42.45	22.7	338.4
$5-4-3$	-	165.63	-	21.8
$5-5-4$	-	28.65	267.3	4.4
$5-7-4$	-	17.18	-	54.7
$5-8-3$	2.881	1.01	4.1	6.6
$6-4-3$	-	94.67	-	29.6
$6-5-3$	-	-	-	2.0
$6-6-3$	5.809	1.20	2.7	2.5
$7-5-3$	-	-	-	28.4
$7-5-5$	-	-	-	0.4

Tabella 9.2: Confronto dei tempi di risoluzione (in secondi) per il problema dei social golfers, al variare dei parametri w, g e s.

Capitolo 10

Conclusioni e lavori futuri

Il lavoro di tesi che è stato descritto verte principalmente su due istanze della programmazione a vincoli su domini finiti:

1. la risoluzione di vincoli su interi
2. la risoluzione di vincoli su insiemi di interi

Per quanto riguarda il punto 1, si è trattato di adattare e migliorare una preesistente implementazione di JSetL. Le modifiche più significative riguardano l'introduzione del concetto di *multi-intervallo* di interi (utilizzato per modellare il dominio delle variabili intere) e delle *euristiche di labeling* (per giungere all'eventuale soluzione attraverso una strategia di ricerca 'informata').

Per quel che concerne i multi-intervalli in letteratura non è stato riscontrato nulla di particolarmente approfondito.
Nel nostro lavoro abbiamo quindi definito formalmente questo dominio, studiandone proprietà e limitazioni.

Per descrivere problemi di soddisfacimento di vincoli su interi, con un approccio analogo alla definizione dei linguaggi del prim'ordine nella logica classica, sono stati introdotti il linguaggio $\mathcal{L}_{\mathcal{FD}}$ ed una sua interpretazione semantica Δ su $\mathbb{Z}$.
Ciò ha permesso di definire formalmente tecniche di risoluzione per CSP basati su tale linguaggio, che in seguito sono state implementate nella libreria JSetL.

La seconda parte della tesi rappresenta invece una novità in JSetL: i vincoli insiemistici vengono risolti tenendo conto del dominio delle variabili

(modellato da un *set-interval*) e della *cardinalità* degli insiemi coinvolti (modellata da una corrispondente variabile intera).
Queste informazione aggiuntive, combinate con le tecniche risolutive introdotte al punto 1, permettono la risoluzione di vincoli insiemistici in modo generalmente più efficiente rispetto al precedente approccio basato su CLP($\mathcal{SET}$).

Simmetricamente a quanto fatto nella prima parte, è stato definito formalmente e analizzato il dominio dei set-interval.
Quindi, sono stati introdotti il linguaggio $\mathcal{L}_{\mathcal{FS}}$, la sua estensione $\mathcal{L}_{\mathcal{FDS}}$ e le rispettive interpretazioni Σ e Φ con un approccio analogo alla definizione del linguaggio $\mathcal{L}_{\mathcal{FD}}$. Ciò ha permesso di formalizzare e implementare strategie per la risoluzione di vincoli su insiemi di interi, avvalendosi delle sopracitate tecniche per la risoluzione di vincoli su interi.

Sulla base del lavoro svolto, i possibili lavori futuri sono molteplici.

Come già osservato, esiste una *corrispondenza biunivoca* tra multi-intervalli ed insiemi di interi appartenenti ad un fissato universo. In alternativa alla rappresentazione 'classica' degli insiemi come collezioni di elementi non ripetuti, potrebbe quindi risultare interessante studiare una rappresentazione basata sui multi-intervalli; essa gode infatti di alcune interessanti proprietà:

- è possibile memorizzare insiemi molto grandi tenendo traccia solamente degli intervalli (o ancora meglio, degli *estremi* degli intervalli) che compongono il corrispondente multi-intervallo.

- il costo delle operazioni insiemistiche tra multi-intervalli non dipende dalla loro cardinalità, bensì dal loro *ordine* (che è sempre minore o uguale alla cardinalità).

- gli intervalli contenuti in un multi-intervallo sono *totalmente ordinati* secondo la relazione $\prec$ definita nella sezione 3.2: questa proprietà può risultare utile per ottimizzare le operazioni fra intervalli.

Il linguaggio $\mathcal{L}_{\mathcal{FD}}$ potrebbe essere inoltre arricchito introducendo nuovi connettivi logici ($\neg$, $\vee$, $\rightarrow$, $\leftrightarrow$), simboli funzionali (meno unario, modulo, divisione intera ecc...) e vincoli globali (su tutti, il vincolo *allDifferent* discusso alla fine del problema 9.1).

Per quanto riguarda il *labeling* sono state definite diverse euristiche per la scelta di valori e variabili. Tuttavia, è possibile implementare altre tecniche. Ad esempio, data una n-pla di variabili $\langle x_1, \ldots, x_n \rangle$ si potrebbe definire una

Variable Choice Heuristic che selezioni la variabile x_i *più vincolata* (cioè, quella che occorre nel maggior numero di vincoli non ancora risolti).
Inoltre, potrebbe risultare interessante (soprattutto per quel che riguarda le strategie randomizzate) imporre un *limite superiore* Ω nella ricerca della soluzione (tale soglia potrebbe essere temporale oppure un numero massimo di choice-points): una volta superato tale Ω, l'utente potrebbe decidere se avviare una nuova ricerca (possibilmente utilizzando altre strategie) oppure fermarsi.

Per quel che concerne i vincoli insiemistici, una possibile estensione potrebbe riguardare lo sviluppo di un risolutore che sappia combinare il trattamento di vincoli su domini finiti, tipicamente efficiente ma limitato a soli insiemi di interi completamente specificati, e il trattamento dei vincoli previsto in CLP($\mathcal{SET}$) e re-implementato in JSetL, in grado di trattare insiemi di oggetti qualsiasi (anche annidati e parzialmente specificati) ma inerentemente più inefficiente.
L'obiettivo sarebbe quello di avere la flessibilità e generalità di CLP($\mathcal{SET}$), utilizzando però le tecniche efficienti di risoluzione su domini finiti ogniqualvolta si ricada nei casi particolari in cui tali tecniche risultino applicabili.

Va inoltre sottolineato che, oltre ai set-interval, esistono altre soluzioni che permettono di rappresentare il dominio di una variabile insiemistica.

Una possibile alternativa, descritta in [10], è rappresentata dai ROBDDs (*Reduced Ordered Binary Decision Diagrams*). Questo tipo di approccio risulta molto performante per alcune classi di CSP contenenti vincoli insiemistici; tuttavia, esso presenta difficoltà nel trattare efficientemente i vincoli di cardinalità.

Una rappresentazione che potrebbe estendere in modo più naturale il lavoro svolto si basa invece su un *ordinamento lessicografico* $\sqsubseteq$ degli insiemi, in alternativa all'*ordinamento parziale* definito dalla relazione $\subseteq$.[9]

L'idea, dati due insiemi X e Y, è che $X \sqsubseteq Y$ se e soltanto se:

$$X = \varnothing \ \vee \ |X| < |Y| \ \vee$$
$$|X| = |Y| \ \wedge \ (min(X) < min(Y) \ \vee \ X \setminus min(X) \sqsubseteq Y \setminus min(Y))$$

Nonostante l'apparente complessità di questa formulazione, intuitivamente si può vedere $\sqsubseteq$ come l'*ordinamento totale* di un reticolo di insiemi scorrendo il corrispondente diagramma di Hasse 'dal basso verso l'alto e da sinistra verso destra'.
Ad esempio, il set-interval $[\varnothing..\{1,2,3\}]$ risulta così ordinato:

$$\varnothing \sqsubseteq \{1\} \sqsubseteq \{2\} \sqsubseteq \{3\} \sqsubseteq \{1,2\} \sqsubseteq \{1,3\} \sqsubseteq \{2,3\} \sqsubseteq \{1,2,3\}$$

In questo modo, è possibile modellare il dominio delle variabili insiemistiche con '*intervalli lessicografici*' del tipo:

$$\langle A..B\rangle \stackrel{def}{=} \{X \subseteq \mathbb{Z} : A \sqsubseteq X \sqsubseteq B\}$$

Questo tipo di rappresentazione risulta molto utile qualora la cardinalità di un insieme assuma un *fissato valore* k: se ad esempio $X :: [\varnothing..[1..10]]$ e $cX = 5$, allora il dominio di X si può ridurre all'intervallo $\langle[1..5]..[5..10]\rangle$ (intuitivamente, 'estraggo' dal reticolo di $2^{10} = 1024$ elementi appartenenti a $[\varnothing..[1..10]]$ solamente la 'riga' corrispondente ai $\binom{10}{5} = 252$ insiemi che hanno cardinalità 5; si noti che utilizzando i set-interval un tale raffinamento non sarebbe possibile, essendo che $[1..5] \not\subseteq [5..10]$ quindi $[[1..5]..[5..10]] = \varnothing$).

Inoltre, l'ordinamento lessicografico permette di rompere in modo naturale le numerose *simmetrie* che tipicamente occorrono nei CSP insiemistici (cfr. Triple di Steiner, Hamming Codes, Social Golfers...).

Queste importanti proprietà comportano una notevole riduzione dello spazio di ricerca: come si è osservato per il problema dei Social Golfers, l'utilizzo di euristiche che ottimizzino la ricerca della soluzione è fondamentale per risolvere efficientemente problemi di una certa complessità.

Una possibile estensione di JSetL potrebbe quindi prevedere l'implementazione di domini 'lessicografici' per le variabili insiemistiche, eventualmente combinando tale rappresentazione con il dominio dei set-interval.
Ad esempio, se il dominio di una variabile X è un set-interval D_X e ad un certo punto della risoluzione la sua cardinalità cX assume un fissato valore k, conviene allora considerare il dominio lessicografico D'_X ottenuto ordinando secondo $\sqsubseteq$ tutti i sottoinsiemi di D_X che hanno cardinalità k.

Inoltre, essendo $\sqsubseteq$ totale, si potrebbe studiare l'utilità di una rappresentazione che permetta di trattare *multi-intervalli lessicografici* in modo analogo a quanto fatto per i multi-intervalli di interi.

Infine, si potrebbero implementare apposite regole e algoritmi per la risoluzione di particolari vincoli quali ad esempio:

- l'*unione generalizzata* di insiemi $\bigcup : \mathcal{P}^2(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ tale che:

$$\bigcup\{X_1, \ldots, X_n\} = X_1 \cup \ldots \cup X_n \qquad \text{per ogni } X, X_1, \ldots, X_n \subseteq \mathbb{Z}$$

- il vincolo di *minimo* su un insieme $min : \mathcal{P}(\mathbb{Z}) \to \mathbb{Z}$ tale che:

$$min(X) \in X \quad \text{e} \quad (\forall\, y \in X)\ y \geq min(X) \qquad \text{per ogni } X \subseteq \mathbb{Z}$$

- il vincolo di *massimo* su un insieme $max : \mathcal{P}(\mathbb{Z}) \to \mathbb{Z}$ tale che:

$$max(X) \in X \quad \text{e} \quad (\forall\ y \in X)\ y \leq max(X) \qquad \text{per ogni } X \subseteq \mathbb{Z}$$

- il vincolo di *somma* su un insieme $sum : \mathcal{P}(\mathbb{Z}) \to \mathbb{Z}$ tale che:

$$sum(X) = \sum_{x \in X} x \qquad \text{per ogni } X \subseteq \mathbb{Z}$$

Bibliografia

[1] Roberto Amadini
Definizione e trattamento del vincolo di cardinalità insiemistica nella libreria JSetL
`http://www.cs.unipr.it/Informatica/Tesi/Roberto_Amadini_20071003.pdf`

[2] Krzysztof R. Apt
Principles of Constraint Programming
Cambridge University Press, 2003.

[3] Francisco Azevedo
Cardinal: A Finite Sets Constraint Solver
Constraints 2007; 12:93-129.

[4] Federico Bergenti, Alessandro Dal Palù, Gianfranco Rossi
Integrating finite domain and set constraints into a set-based constraint language
Fundamenta Informaticae - FUIN, vol. 96, no. 3, pp. 227-252, 2009.

[5] A. Dal Palù, A. Dovier, E. Pontelli, G. Rossi
Integrating finite domain constraints and CLP with sets
ACM Press: New York, 2003; 219-229.

[6] A. Dovier, C. Piazza, E. Pontelli, G. Rossi
Sets and constraint logic programming
ACM TOPLAS 2000; 22(5):861-931.

[7] A. Dovier, E. Pontelli, A. Dal Palù, 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.

[8] Carmen Gervet
Interval Propagation to Reason about Sets: Definition and Implementation

of a Pratical Language
Constraints Journal 1(3), 1997.

[9] Carmen Gervet and Pascal Van Hentenryck
Length-Lex Ordering for Set CSPs
Brown University, Box 1910, Providence, RI 02912

[10] P. J. Hawkins, V. Lagoon and P. J. Stuckey
Solving Set Constraint Satisfaction Problems using ROBDDs
Journal of Artificial Intelligence Research.

[11] Martin Henz, Tobias Müller
An Overview of Finite Domain Constraint Programming
Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies.

[12] Tobias Müller, Martin Müller and Forschungsbereich Programmiersysteme
Finite Set Constraints in Oz
Technische Universität München.

[13] Ulf Nilsson
Logic, Programming and Prolog (Supplement)
`ima.udg.es/Docencia/3105200736/clpfd.pdf`

[14] Daniele Pandini
Progettazione e realizzazione in Java di un risolutore di vincoli su domini finiti
`http://www.cs.unipr.it/Informatica/Tesi/Daniele_Pandini_20080227.pdf`

[15] Francesca Rossi, Peter van Beek, Toby Walsh
Handbook of Constraint Programming (Foundations of Artificial Intelligence)
Elsevier Science Inc. New York, NY, USA.

[16] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

[17] Andrew Sadler and Carmen Gervet
Enhancing set constraint solvers with lexicographic bounds
J Heuristics (2008) 14: 23-67

[18] Christian Schulte and Mats Carlsson
Finite Domain Constraint Programming Systems (2006) Foundations of Artificial Intelligence, 2 (C), pp. 495-526.

[19] Vincent Thornary, Jérôme Gensel, Projet Sherpa
An hybrid representation for set constraint satisfaction problems
Workshop on Set Constraints, Fourth International Conference on Principles and Practice of Constraint Programming, Pisa, Italy, October 26-30, 1998

[20] {log} Home Page
`http://prmat.math.unipr.it/~gianfr/setlog.Home.html`

[21] CSPLib Home Page
`http://www.csplib.org/`

[22] GECODE - An open, free, efficient constraint solving toolkit
`http://www.gecode.org/`

[23] JaCoP - Java Constraint Programming solver
`http://jacop.osolpro.com/`

[24] Java Platform, Standard Edition 6: API Specification
`http://java.sun.com/javase/6/docs/api/`

[25] JSetL Home Page
`http://cmt.math.unipr.it/jsetl.html`

[26] SWI-Prolog Reference Manual
Constraint Logic Programming over Finite Domains
`http://www.swi-prolog.org/man/clpfd.html`