Unione europea
Fondo sociale europeo

MINISTERO DEL LAVORO
E DELLE POLITICHE SOCIALI
Direzione Generale per le Politiche
per l'Orientamento e la Formazione

REGIONE DEL VENETO

Choco solver

Maria Silvia Pini

Dipartimento di Matematica Pura e Applicata

Università di Padova

Choco Solver

- Libreria Java per risolvere problemi di soddisfazione di vincoli
- open-source
- Distribuito sotto una licenza BSD e gestito da sourceforge.net
- Vincitore del“JCP Award for The Most Innovative JSR of 2010”
- Sito web dove reperire il codice e la documentazione
  - http://choco.emn.fr

Schema della presentazione

- Modellare un problema in Choco
  - Definizione delle variabili
  - Definizione dei vincoli
- Risolvere un problema
  - Lettura del modello
  - Definizione di una strategia di ricerca
  - Risoluzione del problema
- Descrizione piu' dettagliata di
  - Variabili
  - Operatori
  - Vincoli

Esempio (1)

- **La matrice magica** (magic square) di ordine n e' un'allocazione di n^2 numeri, solitamente interi distinti, in una matrice tale che
  - la somma di n numeri in tutte le righe, in tutte le colonne e in tutte le diagonali e' uguale alla stessa costante
  - La matrice contiene numeri interi da 1 a n^2 tutti diversi tra loro
  - La somma costante in ogni riga, colonna e diagonale e' detta costante magica M
  - M dipende solo dal valore di n: M(n) = n(n^2+1)/2

- **Esempio**
  - n=3
  - M=3(9+1)/2=15

2	**7**	**6**
9	**5**	**1**
4	**3**	**8**

Esempio (2)

- Dobbiamo risolvere un problema in cui
  - **Il valore delle celle e' sconosciuto**
  - Ogni cella puo' contenere un valore intero compreso tra 1 e n^2
  - Ogni cella deve contenere un valore diverso
  - La somma dei valori di ogni diagonale, di ogni riga e di ogni colonna deve essere uguale alla costante M (M=n(n^2 + 1)
- Vediamo come modellare questo problema in Choco

Modello in Choco

- Per definire il nostro problema, dobbiamo definire un oggetto che si chiama **Model**
- Poiche' vogliamo usare la programmazione con vincoli per risolvere il problema, dobbiamo creare un **CPModel**
- Costanti del problema
  - int n = 3;
  - int M = n * (n * n + 1) / 2;
- Il modello
  - Model m = new CPModel();

Modello: classi da importare

- Per usare questi oggetti dobbiamo importare le seguenti classi
  - import choco.cp.model.CPModel;
  - import choco.kernel.model.Model;
- All'inizio il nostro modello e' vuoto
- Il modello sara' costituito da
  - variabili
  - vincoli (che legano le variabili)

Modello: variabili (1)

- Una variabile e’ un oggetto definito da
  - nome
  - tipo
  - dominio
- Nel nostro esempio le variabili sono le celle sconosciute della matrice

  IntegerVariable cell = Choco.makeIntVar("aCell",1, n*n)
  - aCell e’ una variabile intera
  - Il dominio di aCell e’ definito da 1 a n*n

Modello: variabili (2)

- Nel nostro problema abbiamo bisogno di n^2 variabili che definiamo nel modo seguente

```
IntegerVariable[][] cells = new IntegerVariable[n][n];
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
       cells[i][j] = Choco.makeIntVar("cell" + j, 1, n * n);
       }
   }
```

- Dobbiamo importare le seguenti classi
  - Import choco.kernel.model.variables.integer.IntegerVariable;
  - import choco.Choco;

Modello: vincoli (1)

- Vincoli sulle righe
  - La somma di tutti i valori in ogni riga e' uguale a M
  - Abbiamo bisogno di un operatore somma e di un vincolo di uguaglianza

```
Constraint[] rows = new Constraint[n];
for (int i = 0; i < n; i++) {
rows[i] = Choco.eq(Choco.sum(cells[i]), M);
}
```

  - Dobbiamo importare la seguente classe

```
import choco.kernel.model.constraints.Constraint;
```

  - Dobbiamo aggiungere i vincoli al modello

```
m.addConstraints(rows);
```

Modello: vincoli (2)

- Vincoli sulle colonne
  - Non introduciamo nuove variabili
  - Costruiamo la matrice trasposta di cells

```
IntegerVariable[][] cellsDual = new IntegerVariable[n][n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
     cellsDual[i][j] = cells[j][i];
   }
}
```

  - Dichiariamo i vincoli e li aggiungiamo al modello

```
Constraint[] cols = new Constraint[n];
for (int i = 0; i < n; i++) {
   cols[i] = Choco.eq(Choco.sum(cellsDual[i]), M);
}
m.addConstraints(cols);
```

Modello: vincoli (3)

- Vincoli sulle diagonali
  - Costruiamo la matrice diags a partire da cells

```
IntegerVariable[][] diags = new IntegerVariable[2][n];
for (int i = 0; i < n; i++) {
 diags[0][i] = cells[i][i];
 diags[1][i] = cells[i][(n - 1) - i];
 }
```

  - Aggiungiamo i vincoli direttamente al modello

```
m.addConstraint(Choco.eq(Choco.sum(diags[0]), M));
m.addConstraint(Choco.eq(Choco.sum(diags[1]), M));
```

Modello: vincoli (4)

- Vincoli sulle variabili Alldifferent
  - Il valore di ogni cella deve essere diverso
  - Riordiniamo le variabili inserendole in un array temporaneo

```
IntegerVariable[] allVars = new IntegerVariable[n * n];
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
  allVars[i * n + j] = cells[i][j];
  }
}
m.addConstraint(Choco.allDifferent(allVars));
```

- Il modello e' completato!

Risolutore (1)

- Per risolvere il problema dobbiamo creare un solver

  ```
  Solver s = new CPSolver();
  ```

- Dobbiamo importare le classi

  ```
  import choco.kernel.solver.Solver;
  import choco.cp.solver.CPSolver;
  ```

- Il risolutore deve leggere il modello

  ```
  s.read(m);
  ```

- Ora dobbiamo risolvere il problema

  ```
  s.solve();
  ```

Risolutore (2)

- Stampare i valori della matrice

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    System.out.print(s.getVar(cells[i][j]).getVal() + " ");
   }
 System.out.println();
}
```

- Il codice relativo al risolutore e' completato!
- L'intero programma: MyFirstChocoProgram.java
- Demo di Choco

Esempio semplice

- PROBLEMA
  - Date tre variabili intere x1, x2 e x3 ciascuna con dominio [0,5]
  - trovare un assegnamento di valori a queste variabili che soddisfi i seguenti vincoli
    - x1>x2
    - x1≠x3
    - x2>x3

```
// Build a model
Model m = new CPModel();
// Build enumerated domain variables
IntegerVariable x1 = Choco.makeIntVar("var1", 0, 5);
IntegerVariable x2 = Choco.makeIntVar("var2", 0, 5);
IntegerVariable x3 = Choco.makeIntVar("var3", 0, 5);
```

```
// Build the constraints
Constraint C1 = Choco.gt(x1, x2);
Constraint C2 = Choco.neq(x1, x3);
Constraint C3 = Choco.gt(x2, x3);
// Add the constraints to the Choco model
m.addConstraint(C1);
m.addConstraint(C2);
m.addConstraint(C3);
// Build a solver
Solver s = new CPSolver();
// Read the model
s.read(m);
//Solve
s.solve();
// Print the variable domains
System.out.println("var1 ="  s.getVar(x1).getVal());
System.out.println("var2 =“ + s.getVar(x2).getVal());
System.out.println("var3 =" + s.getVar(x3).getVal());
```

Riassumendo

- Un modello Choco e' definito da
  - Un insieme di variabili ciascuna con un proprio dominio
  - Un insieme di vincoli che legano le variabili
  - Le variabili e i vincoli vanno aggiunti al modello
  - Il modello va legato al risolutore

- Il modello dichiara gli oggetti
- Il risolutore trova il valore di questi oggetti

Variabili

Variabili semplici

- Una variabile e' definita da
  - Tipo
    - Intero (IntegerVariable)
    - Reale (RealVariable)
    - Insieme (SetVariabile)
  - Nome
  - Valori del suo dominio
- Esempio
  - IntegerVariable v1 = Choco.makeIntVar("v1", 1, 3);
- Una o piu' variabili vanno aggiunte al modello
  - model.addVariable(var1);
  - model.addVariables(var2, var3);

Costanti

- Una costante e' una variabile con un dominio contenente un solo valore
- Una integerVariabile con un unico valore e' automaticamente considerata una costante
- Esempio
  - IntegerConstantVariable c10 = Choco.constant(10);
  - RealConstantVariable c0dot0 = Choco.constant(0.0);
  - SetConstantVariable c0_12 = Choco.constant(new int[]{0, 12});
  - SetConstantVariable cEmpty = Choco.emptySet();

Espressioni di variabili ed operatori

- Le variabili si possono combinare in espressioni con appositi operatori
  - IntegerExpressionVariable, RealExpressionVariable
- Esempio
  - IntegerVariable v1 = Choco.makeIntVar("v1", 1, 3);
  - IntegerVariable v2 = Choco.makeIntVar("v2", 1, 3);
  - IntegerExpressionVariable v1pv2 = Choco.plus(v1, v2);
- Operatori
  - Integer: abs, div, ifThenElse, max, min, minus, mod, mult, neg, plus,power, scalar, sum
  - Real: cos, minus, mult, plus, power, sin

Variabili non-decisionali ed obiettivo

- Le variabili aggiunte al modello sono di default decisionali, cioe' incluse nella strategia di ricerca
- Una variabile puo' anche essere definita
  - Variabile non-decisionale
    - model.addVariables(Options.V_NO_DECISION, var5, var6);
  - Variabile obiettivo (solo una variabile puo' essere definita in questo modo)
    - model.addVariable(Options.V_OBJECTIVE, var4);

Variabili di task

- Le variabili di task servono per modellare i problemi di scheduling dove si devono determinare i tempi di inizio e di fine delle attivita'
- Per creare una variabile di task, si possono indicare l'inizio, la fine e la durata prima oppure si indica
  - Il primo tempo di inizio possibile
  - L'ultimo tempo possibile di completamento
  - La durata
- Esempio

```
TaskVariable tvar1 = makeTaskVar("tvar1", 0, 123, 18);
IntegerVariable start = makeIntVar("start", 0, 30);
IntegerVariable end = makeIntVar("end", 10, 60);
TaskVariable tvar2 = makeTaskVar("tvar2", start, end, 10);
```

Vincoli

Vincoli

- Un vincolo
  - riguarda una o piu' variabili del modello
  - specifica delle condizioni che devono essere soddisfatte da queste variabili
- Ogni vincolo deve essere aggiunto al modello
  - model.addConstraint(Choco.neq(var1, var2));
- Aggiungere un vincolo al modello aggiunge automaticamente tutte le variabili al modello
- Una variabile che non e' coinvolta in nessun vincolo non sara' dichiarata nel risolutore durante il passo di lettura del modello

Vincoli semplici e globali

- Vincoli semplici

  ```
  model.addConstraint(Choco.neq(var1, var2));
  ```

- Vincoli globali (esempio: alldifferent)

  ```
  IntegerVariable[] queens = new IntegerVariable[n];
  m.addConstraint(allDifferent(queens));
  ```

  - Appaiono spesso nei problemi reali
  - Gestiti efficientemente, aiutano il solver

Vincoli binari e su insiemi di variabili

- Vincoli binari (coinvolgono solo 2 variabili)
  - eq, geq, gt, leq, lt, neq
    - eq(x; y) stabilisce che x e y devono essere uguali
  - abs, oppositeSign, sameSign
    - abs(x) restituisce il valore assoluto di x

- Vincoli che coinvolgono insiemi di variabili
  - member, notMember
    - member(x;s) stabilisce che x e' un elemento dell'insieme s
  - eqCard, geqCard, leqCard, neqCard
    - eqCard(s; z) stabilisce che la cardinalita' dell'insieme s e' uguale a z

Vincoli di channeling

- L'uso di un modello ridondante e' una tecnica frequente per rafforzare la riduzione dei domini
- I vincoli di channeling legano due modelli dello stesso problema collegando
  - assegnamenti variabile-valore del primo modello e
  - assegnamenti variabile-valore del secondo modello
- Esempi
  - boolChanneling $b_j = 1 \leftrightarrow x = j$
    - Esempio: studente-edizione=1 $\leftrightarrow$ studente=edizione
    - boolChanneling(b; x; v) stabilisce che il valore Booleano b deve essere true se e solo se la variabile x ha valore v
  - domainChanneling $b_j = 1 \leftrightarrow x = j, \forall j,$
  - inverseChanneling $y_j = i \leftrightarrow x_i = j, \forall\ i; j,$

Meta-vincoli logici

- Hanno come argomento dei vincoli
- and, or, implies, ifOnlyIf, ifThenElse, not, nand, nor
  - or(C1;…;Cn) stabilisce che almeno uno dei vincoli C1,…,Cn deve essere soddisfatto
- Esempio

```
IntegerVariable[] vars = new IntegerVariable[n];
Constraint exp = ifOnlyIf( or( eq(x, mult(10, abs(y))), leq(z, 9) ),
alldifferent(vars) );
```

- Variabili e costanti possono essere combinate come ExpressionVariable usando
  - Operatori (es. mult(10,abs(w)))
  - Vincoli semplici (es. leq(z,9))
  - Vincoli globali (es. alldifferent(vars))

Vincoli globali (1)

- Questi vincoli
  - accettano un qualunque numero di variabili
  - hanno degli algoritmi di riduzione dei domini che sono capaci di fare piu' deduzioni che in un modello equivalente senza vincoli globali
- Esempio:
  - alldifferent(a; b; c; d) oppure
  - neq(a,b), neq(a,c), neq(a,d), neq(b,c), neq(b,d), neq(c,d)
    - c,d $\in$ [3; 4]
    - Con il vincolo globale si puo' dedurre che a e b non possono essere istanziate ne' a 3 ne' a 4
    - Questo non puo' essere dedotto dai soli vincoli neq
- C'e' una lunga lista di vincoli globali disponibile in Choco

Vincoli globali (2)

Vincoli sul valore

- Mettono delle restrizioni sui valori che possono essere assegnati ad un insieme di variabili
- allDifferent, atMostNValue, increasingNValue
  - allDifferent(x1,..,xn) stabilisce che i valori di x1,…xn, sono tutti distinti
- among, occurrence, occurrenceMax, occurrenceMin, globalCardinality
  - among(z; (x1,…,xn);s) stabilisce che z e' il numero di xi che appartengono all'insieme s
- nth (element), max, min
  - max(x; z) stabilisce che z e' uguale al piu' grande elemento del vettore x
- sorting, increasingNValue, increasingSum, lex, lexeq, leximin, lexChain, lexChainEq
  - lex((x1,…,xn),(y1,…,yn)) stabilisce un ordinamento lessicografico $x <_{lex} y$

Vincoli globali (3)

- Vincoli di ottimizzazione
  - Vincoli che legano una variabile alla somma dei pesi di una collezione di assegnamenti variabile-valore
  - among, occurrence, occurrenceMax, occurrenceMin, knapsackProblem, equation, costRegular, tree
    - occurrence(IntegerVariable z, IntegerVariable[] x, int v) stabilisce che z e' uguale al numero di elementi in x con valore v
  - globalCardinality, multiCostRegular

Vincoli globali (4)

Vincoli di scheduling

- Vincoli che coinvolgono attivita' che devono essere schedulate nel tempo
- Vincoli temporali: disjoint (tasks) precedence, precedenceDisjoint, precedenceImplied, precedenceReified, forbiddenInterval, tree
  - disjoint(TaskVariable[] t1, TaskVariable[] t2) stabilisce che ogni coppia di task (Ti, Pj) e' in disgiunzione, cioe' i tempi di esecuzione dei due task non si sovrappongono nel tempo
- Vincoli sulle risorse: cumulative, disjunctive, geost
  - cumulative(IntegerVariable[] start, IntegerVariable[] end, IntegerVariable[] duration,int[] height, int capa) stabilisce che un insieme di task definiti dai loro tempi di inizio (start), tempi di fine (end), durate (duration) e consumo (height) e' eseguibile su una risorsa in comune di capacita' limitata (capa)

Solver

Creare il solver

- Per creare il Solver bisogna creare un oggetto CPsolver
  - Solver solver = new CPSolver();
- Una volta definito il modello, il solver lo deve leggere
  - solver.read(model);
- La lettura del modello si divide in
  - Lettura delle variabili
  - Lettura dei vincoli

Lettura delle variabili

- Quando il solver legge le variabili del modello, per ognuna di loro crea una corrispondente variabile nel solver
- Una variabile nel modello
  - e' definita da una rappresentazione astratta del suo dominio iniziale
- Una variabile nel solver
  - racchiude al proprio interno la rappresentazione concreta del suo dominio
  - Mantiene lo stato corrente del dominio durante la ricerca
  - Pertanto non si puo' accedere al valore di una variabile dal modello, ma dalla corrispondente variabile del solver

```
IntegerVariable x = Choco.makeEnumIntVar("x", 1, 100); // model variable
IntDomainVar xOnSolver = solver.getVar(x); // solver variable
```

Variabili nel solver

- Si puo' vedere lo stato di una IntDomainVar usando i seguenti metodi pubblici
  - getInf() ritorna il lower bound corrente della variabile
  - getSup() ritorna l'upper bound corrente della variabile
  - getVal() ritorna il valore della variabile, se e' correntemente istanziata
  - isInstantiated() verifica se il dominio e' correntemente ridotto ad un unico valore
  - canBeInstantiatedTo(int v) checks verifica se il valore v appartiene correntemente al dominio della variabile
  - getDomainSize() restituisce la taglia corrente del dominio

Lettura dei vincoli

- Per ogni vincolo del modello viene generato un corrispondente vincolo nel solver
- Ogni vincolo nel solver incapsula un algoritmo di riduzione dei domini che e' chiamato durante la ricerca quando
  - Si verifica un evento esterno (es., rimozione di un valore o modifica di un bound) su qualche variabile del vincolo

Risolvere un problema

- Ci sono vari metodi per risolvere un problema
- Tutti questi metodi restituiscono uno di questi valori
  - Boolean.TRUE se almeno una soluzione e' stata trovata
  - Boolean.FALSE se si e' provato che il problema non ammette soluzioni
  - Null quando si e' raggiunto uno dei limiti della ricerca

Metodi del solver (1)

- solve() or solve(false): esegue la ricerca con backtracking finche'
  - Trova la prima soluzione (restituisce Boolean.TRUE)
  - Prova che nessuna soluzione esiste (restituisce Boolean.FALSE)
  - Raggiunge un limite di ricerca (restituisce null)
- nextSolution():
  - si puo' chiamare solo dopo che una chiamata di solve() o nextSolution() hanno restituito Boolean.TRUE
  - Esegue la ricerca a partire dalla foglia soluzione raggiunta dalla precedente chiamata di solve() o nextSolution() finche'
    - trova una nuova soluzione (restituisce Boolean.TRUE)
    - prova che non esiste nessuna soluzione (restituisce Boolean.FALSE)
    - raggiunge un limite di ricerca (restituisce null).

Metodi del solver (2)

- solveAll() or solve(true): esegue la ricerca finche'
  - Trova tutte le soluzioni (restituisce Boolean.TRUE)
  - Prova che nessuna soluzione esiste (restituisce Boolean.FALSE)
  - Raggiunge un limite di ricerca (restituisce Boolean.TRUE se trova almeno una prima soluzione e null altrimenti)

Strategia di ricerca (1)

- E’ un elemento chiave di ogni approccio di programmazione con vincoli
- Negli approcci di backtracking o branch-and-bound
  - La ricerca e’ organizzata come un albero dove
    - ogni nodo corrisponde ad un sottoalbero della ricerca
    - ogni nodo figlio e’ una suddivisione dello spazio di ricerca del suo nodo padre
  - L’albero viene visitato progressivamente applicando delle strategie di branching che determinano
    - come suddividere lo spazio di ricerca di ogni nodo
    - In che ordine esplorare i nodi figli
- Vedremo come definire la nostra strategia di ricerca in Choco

Strategia di ricerca (2)

- Gli approcci backtracking e branch-and-bound **visitano l'albero di ricerca usando Depth-First Search**
  - Istanziano una variabile e fanno propagazione (cioe' riducono i domini delle variabili non ancora istanziate)
  - Se il dominio di qualche variabile non-istanziata diventa vuoto, fanno backtrack (cioe' valutano il prossimo nodo nell'albero)
  - Altrimenti dividono lo spazio di ricerca e valutano il primo nodo figlio
  - Per dividere lo spazio di ricerca
    - Assegnano una variabile x ad un valore v (cioe' $x=v$) e
    - Proibiscono questo assegnamento (cioe' $x \neq v$)
- Choco fornisce
  - questa strategia di branching
  - Metodi per costruire euristiche per selezionare variabile e valore all'interno di questa strategia

Metodi per manipolare il solver

- propagate():
  - Lancia la riduzione dei domini delle variabili non ancora istanziate
  - Lancia una ContradictionException quando il dominio di una variabile diventa vuoto
  - Questo metodo viene chiamato ad ogni nodo dell'albero di ricerca costruito dai metodi citati sopra
- isFeasible(): true se il solver ha gia' trovato almeno una soluzione

Altri metodi del solver (1)

- maximize(Var obj, boolean restart),
  maximize(boolean restart): esegue la ricerca finche'
  - Trova una soluzione che massimizza la funzione obiettivo obj (restituisce Boolean.TRUE)
  - Prova che nessuna soluzione esiste (restituisce Boolean.FALSE)
  - Raggiunge un limite di ricerca (restituisce Boolean.TRUE se trova almeno una prima soluzione e null altrimenti)
  - Strategia utilizzata: ogni volta che una soluzione viene trovata ad una foglia dell'albero di ricerca, **ricerca un'altra soluzione con un valore della funzione obiettivo piu' grande**, finche' prova che non esiste nessun miglioramento

Altri metodi del solver (2)

- Il parametro restart e' un valore Booleano che indica
  - se la ricerca continua dalla soluzione foglia con un backtrack (se settato a FALSE) oppure
  - se rilancio la soluzione dalla radice (se settato a TRUE)
- I restart
  - Interrompono la ricerca nell'albero corrente e iniziano la ricerca in un **nuovo albero** a partire dal nodo radice
  - Hanno senso solo quando vengono usati in contemporanea con strategie di branching dinamiche randomizzate che ci assicurano di non costruire lo stesso albero di ricerca due volte. Per esempio
    - **DomOverWDegBranchingNew**
    - **DomOverWDegBinBranchingNew**

Una strategia di ricerca randomizzata

- DomOverWDegBranchingNew
  - E' una strategia di branching n-aria che assegna valori distinti ad una variabile intera
  - Ad ogni variabile sono associati tre valori
    - Dom: taglia del dominio corrente
    - Deg: numero corrente dei vincoli che coinvolgono quella variabile non ancora istanziati
    - W: somma dei costi associati con questi Deg vincoli
  - La strategia seleziona la variabile con
    - **piu' piccola ratio r = Dom/W*Deg**
  - Le tie sono rotte in modo random

    DomOverWDegBranchingNew(Solver s, IntDomainVar[] vars, ValIterator valIt, Number seed)

Visualizzare il risultato della ricerca (1)

```
import choco.kernel.common.logging.ChocoLogging;
// Before the resolution
ChocoLogging.toVerbose();
//... resolution declaration
solver.solve();
// And after the resolution
ChocoLogging.flushLogs();
```

- Ricerca completata con almeno una soluzione
  - ** CHOCO : Constraint Programming Solver
  - ** CHOCO v2.1.1 (May, 2011), Copyleft (c) 1999-2011
  - - Search completed
  - Solutions: 1
  - Time (ms): 16
  - Nodes: 4
  - Backtracks: 5
  - Restarts: 0

Visualizzare il risultato della ricerca (2)

- Ricerca incompleta perche' si e' raggiunto il limite di tempo
  - Limit : {0} ,
  - [Maximize : {1} ,]
  - [Minimize : {2} ,]
  - Solutions : {3} ,
  - Times (ms) : {4} ,
  - Nodes : {5} ,
  - Backtracks : {6} ,
  - Restarts : {7}.
    - Le parentesi quadre [linea] indicano che la linea e' opzionale
    - Maximize (resp. Minimize) indica il valore migliore della funzione obiettivo prima di bloccare la ricerca
- Visualizzare tutte le informazioni della ricerca

  ChocoLogging.*setVerbosity(Verbosity.FINEST);*

Definire strategie di branching (1)

- Le strategie di branching si applicano alle variabili del solver
- Le euristiche di selezione della variabile e del valore possono essere definite separatamente
- Nella strategia di branching **AssignVar**
  - Prima viene selezionata la variabile
  - Poi viene selezionato il valore della variabile
- Esempio

  new AssignVar(new MinDomain(solver), new **IncreasingDomain()**);

  new AssignVar(new MinDomain(solver), new **MinVal()**)
  - selezioniamo una variabile con dominio di taglia minima
  - In ogni branch le assegnamo uno dei valori del suo dominio, selezionato in ordine crescente
  - Questa strategia e' pre-definita
    BranchingFactory.**minDomMinVal**(solver);

Definire strategie di branching (2)

- La scelta della variabile puo' richiedere **calcoli specifici** prima e dopo il branching
- In questo caso l'euristica di selezione della variabile puo' essere implementato **direttamente nell'oggetto branching strategy**
- Esempio: **DomOverWDegBranchingNew**
  - Per ogni variabile sono associati tre valori
    - Dom: taglia del dominio corrente
    - Deg: numero corrente dei vincoli che coinvolgono quella variabile non ancora istanziati
    - W: somma dei costi associati con questi Deg vincoli
  - La strategia seleziona la variabile con
    - **piu' piccola ratio r = Dom/W*Deg**
  - Le tie sono rotte in modo random

Definire strategie di branching (3)

- E’ possibile definire strategie di branching diverse per insiemi di variabili diverse
- Una strategia di branching deve essere definita come un goal

  void addGoal(AbstractIntBranchingStrategy branching);
- Questo metodo va chiamato sull’oggetto solver **prima** di chiamare il metodo di soluzione
- La lista iniziale dei goal e’ vuota
- I goal vengono inseriti nell’ordine in cui sono stati dichiarati
- Se si vuole rilanciare la ricerca, la lista dei goal viene resettata con il metodo solver.clearGoals();

Strategie di default

- Si applicano a tutte le variabili decisionali e dipendono dal tipo delle variabili
- Set: AssignSetVar, MinDomainSet
- Integer: DomOverWDegBranchingNew, IncreasingDomain
- Real: AssignInterval, CyclicRealVarSelector, RealIncreasingDomain

Esempio (1)

- Aggiungiamo quattro oggetti di branching al solver s
  - s.addGoal(BranchingFactory.minDomMinVal(s, s.getVar(vars1)));
    - Strategia AssignVar applicata alle variabili var1
  - s.addGoal(new AssignVar(new DomOverDeg(s, s.getVar(vars2)), new DecreasingDomain()));
    - Altra strategia AssignVar applicata alle variabili var2
  - s.addGoal(new AssignSetVar(new MinDomSet(s, s.getVar(svars)), new MinEnv()));
    - strategia AssignSetVar applicata all'insieme delle variabili svars
  - s.addGoal(BranchingFactory.randomIntSearch(s, seed));
    - Strategia random che si applica a tutte le variabili decisionali del solver
  - s.solve();

Esempio (2)

- s.addGoal(BranchingFactory.minDomMinVal(s, s.getVar(vars1)));
- s.addGoal(new AssignVar(new DomOverDeg(s, s.getVar(vars2)), new DecreasingDomain()));
- s.addGoal(new AssignSetVar(new MinDomSet(s, s.getVar(svars)), new MinEnv()));
- s.addGoal(BranchingFactory.randomIntSearch(s, seed));
  s.solve();

- La strategia di branching
  - Prima considera le variabili vars1 finche' sono tutte istanziate
  - Poi considera le variabili var2 e dopo le variabili svars
  - Infine applica una strategia di ricerca random a tutte le variabili intere diverse da vars1 e vars2 che non sono ancora state istanziate

Strategie di branching pre-definite

- Le strategie di branching definite in Choco
  - AssignInterval, AssignOrForbidIntVarVal, AssignOrForbidIntVarValPair
    - AssignOrForbidIntVarVal e' una strategia di branching binaria che assegna un valore ad una variabile intera
      - Nel primo ramo il valore selezionato viene assegnato alla variabile
      - Nel secondo ramo il valore e' tolto dal dominio della variabile
  - AssignSetVar, AssignVar, DomOverWDegBranchingNew, DomOverWDegBinBranchingNew,
  - ImpactBasedBranching, PackDynRemovals, SetTimes, TaskOverWDegBinBranching

Selettori di variabili pre-definiti

- Le strategie di selezione della variabile da istanziare
  - CompositeIntVarSelector, LexIntVarSelector, MaxDomain, MaxRegret,
  - MaxValueDomain, MinDomain, MinValueDomain, MostConstrained
    - MinDomain: assegna la variabile con dominio di taglia minima
  - RandomIntVarSelector, StaticVarOrder, MaxDomSet, MaxRegretSet, MaxValueDomSet, MinDomSet, MinValueDomSet,
  - MostConstrainedSet, RandomSetVarSelector, StaticSetVarOrder
    - MostConstrainedSet: assegna l'insieme delle variabili coinvolte nel piu' grande numero di vincoli inizialmente presenti nel solver

Selettori di valori pre-definiti

- Le strategie di selezione del valore da assegnare alla variabile
  - MaxVal, MidVal, MinVal
    - MaxVal: seleziona il piu' grande valore nel dominio della variabile
  - BestFit, CostRegularValSelector, FCostRegularValSelector, RandomIntValSelector
    - RandomIntValSelector: seleziona un valore random nel dominio della variabile intera
    - Esempio

```
Model m = new CPModel();
……
Solver s = new CPSolver();
s.read(m);
s.setValIntSelector(new RandomIntValSelector(seed));
s.setVarIntSelector(new RandomIntVarSelector(s, seed));
s.solveAll()
```

Restart (1)

- I restart bloccano l'albero di ricerca corrente e costruiscono un nuovo albero di ricerca dal nodo radice
- Sono una buona strategia di ricerca quando si vuole ottimizzare un problema NP-hard in un tempo limitato

  setGeometricRestart(int base, double grow);

  setGeometricRestart(int base, double grow, int restartLimit);

  - base: indica il numero di backtrack massimo permesso nel primo albero di ricerca
  - Una volta che tale limite e' raggiunto
    - Parte un restart
    - La ricerca continua finche' sono stati fatti base*grow backtrack
    - Dopo ogni restart il numero massimo di backtrack possibili e' aumentato del fattore grow.
    - restartLimit: indica il numero massimo di restart

Restart (2)

- Esempio

```
CPSolver s = new CPSolver();
s.read(model);
s.setGeometricRestart(14, 1.5d);
s.setFirstSolution(true);
s.generateSearchStrategy();
s.attachGoal(new DomOverWDegBranching(s, new
   IncreasingDomain()));
s.launch();
```

Limiti sullo spazio di ricerca (1)

- Il solver permette di imporre limiti allo spazio di ricerca secondo vari criteri
- Quando il solver raggiunge uno di questi limiti la ricerca si ferma anche se non ha trovato soluzione
- time limit
  - Riguarda il tempo trascorso dall'inizio della ricerca
  - E' settato usando setTimeLimit(int timeLimit), dove l'unita' di misura e' il millisecondo
  - getTimeCount() restituisce il tempo totale di esecuzione
- node limit
  - Riguarda il numero di nodi esplorati
  - E' settato usando setNodeLimit(int nodeLimit)
  - getNodeCount() restituisce il numero totale dei nodi esplorati

Limiti sullo spazio di ricerca (2)

- backtrack limit
  - Riguarda il numero di backtrack fatti
  - E’ settato usando setBackTrackLimit(int backtrackLimit)
  - getBackTrackCount() restituisce il numero totale dei backtrack
- fail limit
  - Riguarda il numero di fallimenti
  - Un fallimento e’ settato usando setFailLimit(int failLimit)
  - getFailCount() restituisce il numero totale dei fallimenti

Unione europea
Fondo sociale europeo

MINISTERO DEL LAVORO
E DELLE POLITICHE SOCIALI
Direzione Generale per le Politiche
per l'Orientamento e la Formazione

REGIONE DEL VENETO

Choco solver

Maria Silvia Pini

Dipartimento di Matematica Pura e Applicata

Università di Padova

Grazie per l'attenzione!