Progetto di Programmazione Dichiarativa

Problema delle matrici non sovrapponibili

Federico Corò

23 febbraio 2016

Testo progetto

Input: quattro numeri interi N, M, Q, K
Problema: dire se esistono K matrici di dimensione $N \times M$ (N righe M colonne) che abbiano come elementi numeri interi nell'insieme $\{1, 2, 3, 4, \ldots, Q\}$. Le K matrici devono essere a due a due non sovrapponibili. Due matrici sono sovrapponibili se possono essere anche parzialmente sovrapposte.

Esempio Per esempio, con $N = 3, M = 5, Q = 5$:

$$\begin{array}{ccccc} 4 & 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 1 & 5 \\ 1 & 1 & 1 & 2 & 2 \end{array}$$

è sovrapponibile con

$$\begin{array}{ccccc} 3 & 1 & 5 & 5 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 3 & 4 & 4 \end{array}$$

infatti possono essere scritte in questo modo:

$$\begin{array}{ccccccc} 4 & 1 & 2 & 3 & 4 & & \\ 2 & 2 & 3 & 1 & 5 & 5 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 \\ & & 1 & 2 & 3 & 4 & 4 \end{array}$$

La sovrapposizione può avvenire su qualsiasi porzione delle matrici, anche su un "lato" o solo su un "angolo" della matrice.
Per esempio la prima matrice e' sovrapponibile con la matrice

$$\begin{array}{ccccc} 2 & 4 & 2 & 3 & 2 \\ 1 & 2 & 2 & 2 & 1 \\ 2 & 1 & 3 & 4 & 1 \end{array}$$

infatti possono essere scritte in questo modo:

4	1	2	3	4				
2	2	3	1	5				
1	1	1	2	2	4	2	3	2
				1	2	2	2	1
				2	1	3	4	1

Implementazione

Sono state scritte due soluzioni al problema in ASP (*progetto.lp* e *progetto2.lp*), entrambe sono state testate utilizzando *gringo* e *clasp*[1].

L'approccio utilizzato in entrambi i progetti è di *generate and test*, si ha quindi una parte in comune, quella iniziale, il cui scopo è quello di arrivare a generare tutti le possibili matrici dati gli input sopra citati del problema.

```
nRows(3).
nCols(3).
setQ(3).
valK(4).

rows(1..X) :- nRows(X).
cols(1..X) :- nCols(X).
values(1..X) :- setQ(X).
valuesID(1..M) :- valK(M).

1 { m(T, Row, Col, Val) : values(Val)} 1 :- rows(Row), cols(Col),
    valuesID(T).
```

Si hanno quindi nelle prime quattro righe, i fatti che identificano rispettivamente N, M, Q, K.
Seguiti da quattro regole che servono a creare rispettivamente i fatti riga, colonna, *values* identifica i possibili valore che possono essere scritti nelle matrici e infine *valuesID* identifica gli id delle M matrici cercate. Questo poichè ci serve distinguere le diverse matrici per confrontarle.
Infine viene utilizzato un *ground cardinality constraint* per la generazione di tutte le possibili matrici, verranno creati infatti tanti fatti del tipo *m(T, Row, Col, Val)*, che identificherà rispettivamente: la matrice con id T nel punto Row, Col contiene il valore Val.

Prima implementazione

La prima implementazione si riferisce al file *progetto.lp*.
L'idea alla base di questa prima implementazione è quella di cercare, date due matrici, due sottomatrici uguali, quindi tramite un vincolo eliminare questa possibilità dalle soluzioni cercate.
Il seguente codice ha proprio questo scopo, in questo specifico caso vengono

[1] gringo version 4.2.1, clasp version 2.1.4

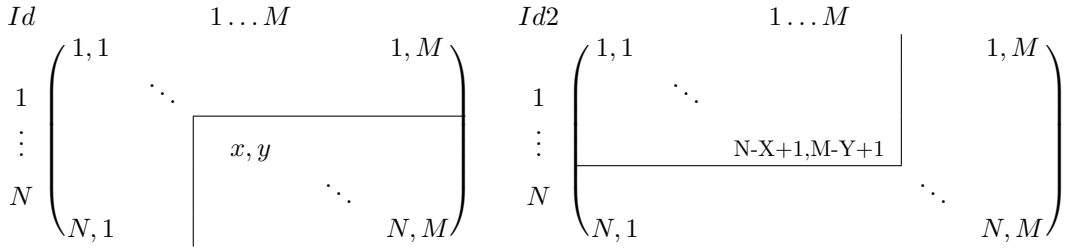
cercate matrici che si espandono verso destra, verrà ripetuto poi all'interno del progetto un uguale frazione di codice per le sottomatrici che si espandono verso sinistra.

```
checkSubMatrixAUX(Id, Id2, X, Y, Z, W) :- rows(X), cols(Y), rows(Z)
    , cols(W), m(Id, X, Y, Val), m(Id2, Z, W, Val), Id != Id2,
    checkSubMatrixAUX(Id, Id2, X+1, Y, Z+1, W), checkSubMatrixAUX(
    Id, Id2, X, Y+1, Z, W+1), checkSubMatrixAUX(Id, Id2, X+1, Y+1,
    Z+1, W+1).

checkSubMatrix(Id, Id2) :- valuesID(Id), valuesID(Id2), Id != Id2,
    rows(X), cols(Y), checkSubMatrixAUX(Id, Id2, X, Y, 1, 1).

:- checkSubMatrix(Id, Id2), valuesID(Id), valuesID(Id2), Id != Id2.
```

Ovvero l'idea è: "fissato" un punto X, Y nella matrice Id, si controlla punto a punto la sottomatrice la cui diagonale termina in N, M con la sottomatrice che inzia dal punto 1,1 della matrice $Id2$. La seguente immagine dovrebbe chiarire meglio.



Durante il *grounding* verranno quindi creati tutti i possibili atomi per tutte le possibili X e Y, con l'atomo *checkSubMatrixAUX* che è vincolato ad essere vero se è sono veri i suoi "vicini".
Questo, grazie al vincolo, porterà ad eliminare tutte le matrici che hanno una sottomatrice uguale.

Di seguito l'output del programma dato il seguente input: $N = 3, M = 3, Q = 3, M = 4$. Per aiutare la leggibilità dell'output è stato utilizzato *#show m/4.*

```
m(1,1,1,1) m(2,3,3,3) m(3,3,3,3) m(4,3,3,3) m(2,1,1,2) m(1,3,3,3)
m(3,1,1,2) m(4,1,1,1) m(1,1,2,3) m(2,1,2,3) m(3,1,2,3) m(4,1,2,3)
m(1,1,3,1) m(2,3,1,3) m(3,3,1,3) m(4,3,1,3) m(2,1,3,2) m(1,3,1,3)
m(3,1,3,2) m(4,1,3,1) m(2,3,2,3) m(3,3,2,3) m(4,3,2,3) m(1,3,2,3)
m(1,2,1,2) m(2,2,1,1) m(3,2,1,2) m(4,2,1,1) m(1,2,2,3) m(2,2,2,3)
m(3,2,2,3) m(4,2,2,3) m(1,2,3,2) m(2,2,3,1) m(3,2,3,2) m(4,2,3,1)
```

I punti sopra descritti formerebbero le seguenti matrici:

$$1: \begin{pmatrix} 1 & 3 & 1 \\ 2 & 3 & 2 \\ 3 & 3 & 3 \end{pmatrix} \quad 2: \begin{pmatrix} 2 & 3 & 2 \\ 1 & 3 & 1 \\ 3 & 3 & 3 \end{pmatrix} \quad 3: \begin{pmatrix} 2 & 3 & 2 \\ 2 & 3 & 2 \\ 3 & 3 & 3 \end{pmatrix} \quad 4: \begin{pmatrix} 1 & 3 & 1 \\ 1 & 3 & 1 \\ 3 & 3 & 3 \end{pmatrix}$$

Seconda implementazione

Per la seconda implementazione (file *progetto2.lp*) si è cercata una soluzione più "leggera", in numero di atomi creati durante il grounding, rispetto alla precedente.

```
checkSubMatrixSXAUX(Id, Id2, X, Y) :- rows(A), cols(B), X <= A, Y
    <= B, rows(X), cols(Y), valuesID(Id), valuesID(Id2), Id != Id2,
     m(Id, A, B, Val), m(Id2, A-X+1, B-Y+1, Val2), Val != Val2.

checkSubMatrixSX(Id, Id2) :- valuesID(Id), valuesID(Id2), Id != Id2
    , rows(X), cols(Y), not checkSubMatrixSXAUX(Id, Id2, X, Y).

:- checkSubMatrixSX(Id, Id2), valuesID(Id), valuesID(Id2), Id !=
    Id2.
```

In questo secondo caso si è creato una regola *checkSubMatrixSXAUX* che "fissato" un punto X, Y nella matrice Id rende vera la testa solo se esiste almeno un punto nella sottomatrice che sia diverso (confrontandolo sempre con la sottomatrice della matrice $Id2$). Questo poichè basta un solo punto diverso nella sottomatrice per renderle non sovrapponibili.

Nella regola *checkSubMatrixSX* si richiede quindi nel corpo che non sia vera la regola precedente, questo per poter modellare il fatto che le due matrici $Id, Id2$ siano effettivamente sovrapponibili, così infine da poter mettere questo atomo sul vincolo ed eliminare tutte le matrici che a noi non servono.
Di seguito l'output del programma dato il seguente input: $N = 2, M = 3, Q = 2, M = 2$. Questa volta poichè si creano meno atomi durante il grounding possiamo far vedere l'intero output, ma sempre per matrici piccole.

```
nRows(2) nCols(3) setQ(2) valK(2) rows(1) rows(2) cols(1) cols(2)
    cols(3) values(1) values(2) valuesID(1) valuesID(2)
    checkSubMatrixSXAUX(1,2,1,1) m(1,1,1,1) m(2,1,1,2)
    checkSubMatrixSXAUX(1,2,1,2) m(1,1,2,2) m(2,1,2,1)
    checkSubMatrixSXAUX(1,2,1,3) m(1,1,3,1) m(2,1,3,2)
    checkSubMatrixSXAUX(2,1,1,1) checkSubMatrixSXAUX(2,1,1,2)
    checkSubMatrixSXAUX(2,1,1,3) checkSubMatrixSXAUX(1,2,2,1) m
    (1,2,1,1) m(2,2,1,2) checkSubMatrixSXAUX(1,2,2,2) m(1,2,2,1) m
    (2,2,2,2) checkSubMatrixSXAUX(1,2,2,3) m(1,2,3,1) m(2,2,3,2)
    checkSubMatrixSXAUX(2,1,2,1) checkSubMatrixSXAUX(2,1,2,2)
    checkSubMatrixSXAUX(2,1,2,3) checkSubMatrixDXAUX(1,2,1,3)
    checkSubMatrixDXAUX(1,2,1,2) checkSubMatrixDXAUX(1,2,1,1)
    checkSubMatrixDXAUX(2,1,1,3) checkSubMatrixDXAUX(2,1,1,2)
    checkSubMatrixDXAUX(2,1,1,1) checkSubMatrixDXAUX(1,2,2,3)
    checkSubMatrixDXAUX(1,2,2,2) checkSubMatrixDXAUX(1,2,2,1)
    checkSubMatrixDXAUX(2,1,2,3) checkSubMatrixDXAUX(2,1,2,2)
    checkSubMatrixDXAUX(2,1,2,1)
```

Le cui matrici ricostruite saranno:

$$1 : \begin{pmatrix} 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad 2 : \begin{pmatrix} 2 & 1 & 2 \\ 2 & 2 & 2 \end{pmatrix}$$