Andrea Bianchini

# UN GENERATORE DI SCHEMI SUDOKU IN JAVA

“Se il settore dell’automobile si fosse sviluppato come l’industria informatica, oggi avremmo veicoli che costano 25 dollari e fanno 500 Km con un litro.”

BILL GATES

Prefazione

Java a tutt'oggi è un linguaggio di programmazione adottato dalla stragrande maggioranza di scuole ed università per l'insegnamento della disciplina della programmazione. E' un linguaggio molto diffuso ed ha la caratteristica di essere indipendente dalla piattaforma in cui viene eseguito oltre ovviamente a fornire un robusto supporto alla programmazione orientata agli oggetti.
Dotato di un costrutto coerente e semplice, Java è l'ideale per affrontare i problemi da un punto di vista didattico.
In questo libro presento una applicazione da me scritta in Java che utilizza una libreria per la soluzione dell'algoritmo del simplesso scritta dal Dott. Stefano Scarioli. Questa applicazione è in grado di generare uno schema Sudoku completo e corretto ogni volta diverso, ho calcolato che i diversi possibili schemi generabili da questa applicazione sono oltre 360.000, ma, modificando di poco l'algoritmo, si potrebbe arrivare ad oltre 130 miliardi ed oltre di schemi diversi.

Andrea Bianchini

“Il **sudoku** (giapponese: 数独, *sūdoku*, nome completo **数字は独身に限る *Sū****ji*** *wa* ***doku****shin* *ni kagiru*, che in italiano vuol dire "sono consentiti solo numeri solitari") è un gioco di logica nel quale al giocatore o solutore 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 9 righe orizzontali, 9 colonne verticali e in 9 "sottogriglie" di 3×3 celle contigue. Queste sottogriglie sono delimitate da bordi in neretto e chiamate *regioni*. Le griglie proposte al giocatore hanno da 20 a 35 celle contenenti un numero. Lo scopo del gioco è quello di riempire le caselle bianche con numeri da 1 a 9 in modo tale che in ogni riga, in ogni colonna e in ogni regione siano presenti tutte le cifre da 1 a 9, quindi senza ripetizioni. In tal senso lo schema, una volta riempito correttamente, appare come un quadrato latino.

Il gioco fu inventato dal matematico svizzero Eulero da Basilea (1707-1783). La versione moderna del gioco fu pubblicata per la prima volta nel 1979 dall'architetto statunitense Howard Garns all'interno del Dell Magazines con il titolo "Number Place". In seguito fu diffuso in Giappone dalla casa editrice Nikoli nel 1984, per poi diventare noto a livello internazionale soltanto a partire dal 2005.

I primi giochi di logica basati sui numeri apparvero sui giornali verso la fine del XIX secolo, quando alcuni enigmisti francesi iniziarono a sperimentarli rimuovendo opportunamente dei numeri dai quadrati magici. *Le*

*Siècle*, un quotidiano parigino, pubblicò nel 1892 un quadrato magico di dimensioni 9×9 parzialmente completo con sottoquadrati di dimensioni 3×3. Non si trattava di un sudoku così come lo conosciamo oggi poiché conteneva numeri a doppia cifra e, per essere risolto, richiedeva l'aritmetica piuttosto che la logica, ma ammetteva comunque la regola per cui ogni riga, colonna e sottoquadrato dovesse contenere gli stessi numeri senza ripeterli. Successivamente un giornale rivale de *Le Siècle*, *La France*, ridefinì le regole di questo gioco, avvicinandosi di molto al sudoku moderno: ogni riga, colonna e sottoquadrato del quadrato magico doveva essere riempita soltanto con i numeri da 1 a 9, sebbene i sottoquadrati non fossero marcati all'interno dello schema. Questi giochi settimanali furono pubblicati anche da altri quotidiani francesi come *L'Echo de Paris* per circa un decennio, ma poi scomparvero all'epoca della Prima guerra mondiale.

Secondo l'enigmista statunitense Will Shortz, il sudoku moderno fu realizzato da Howard Garns, un ex architetto in pensione dell'Indiana(morto nel 1989), e pubblicato per la prima volta nel 1979 da Dell Magazines all'interno della rivista *Dell Pencil Puzzles and Word Games* con il titolo *Number Place*.

Il gioco venne introdotto in Giappone dalla casa editrice Nikoli nella rivista *Monthly Nikolist* nell'aprile del 1984 con il titolo *Suuji wa dokushin ni kagiru* (数字は独身に限る[?]), successivamente abbreviato da Maki Kaji in *Sudoku* prelevando soltanto i primi caratteri kanji del

nome completo. Nel 1986 Nikoli introdusse due novità: il numero massimo di celle già riempite fu ristretto a 32 e le griglie diventarono "simmetriche" (nel senso che i numeri già stampati venivano distribuiti su celle simmetriche).

Nell'ottobre del 2004 il sudoku venne importato in Gran Bretagna da un ex giudice neozelandese, Wayne Gould, per poi diffondersi in Europa e nel resto del mondo nel 2005.

## Descrizione matematica

Come tutti i giochi logici, Sudoku può essere descritto completamente mediante nozioni di logica; in questo caso si applica la combinatoria.

Il gioco si svolge in matrici, che chiamiamo *matrici Sudoku* di aspetto 9×9 (le griglie) le cui caselle possono contenere un elemento di un insieme di 9 oggetti distinguibili, oppure un ulteriore oggetto diverso dai precedenti. Per descriverle conveniamo che le righe e le colonne delle matrici siano individuate dagli interi da 1 a 9, che i nove oggetti siano gli interi dell'insieme 9 := {1, 2, 3, 4, 5, 6, 7, 8, 9}, che l'oggetto ulteriore sia denotato con la lettera b e che una casella contenente b sia detta casella bianca o vuota. Una matrice Sudoku *M* viene considerata suddivisa in 9 blocchi di aspetto 3×3 che denotiamo $B_{h,k}$ con h,k = 1, 2, 3; il blocco $B_{h,k}$ riguarda, per la matrice *M*, le righe relative agli indici 3h-2, 3h-1 e 3h e le colonne relative agli indici 3k-2, 3k-1 e 3k. In ogni

riga, colonna e regione di una matrice Sudoku i valori interi non possono essere ripetuti.

Una *istanza di Sudoku*, detta anche *griglia proposta* o *matrice incompleta*, è una matrice Sudoku che presenta alcune celle bianche. Scopo del gioco è la trasformazione della griglia proposta in una *matrice completa*, cioè in una matrice priva di celle bianche e quindi tale che in ogni sua riga, colonna e regione compaiano tutti gli elementi di 9 (ciascuno una sola volta). Si osserva che una matrice Sudoku completa è un quadrato latino di ordine 9 avente per blocchi matrici 3×3 con i nove numeri da 1 a 9.

Affinché una matrice incompleta sia considerata valida, ai fini del gioco, è necessario che la soluzione sia univoca, ovvero non devono sussistere due o più soluzioni differenti, nei quali casi il gioco viene considerato non valido. Nei casi di varianti di sudoku (per esempio *killer*, *jigsaw*, *x*, *toroidale*, ecc.) ulteriori condizioni devono essere verificate affinché la matrice risulti valida. La difficoltà di un sudoku non è data dalla quantità di numeri iniziali, bensì dalla loro disposizione.

Le soluzioni di una qualsiasi altra matrice incompleta sono un sottoinsieme delle soluzioni della matrice vuota.

Il numero delle soluzioni del Sudoku classico è 6.670.903.752.021.072.936.960, approssimativamente $6{,}67 \cdot 10^{21}$. Il numero delle soluzioni sostanzialmente diverse escludendo le simmetrie dovute

a rotazioni, riflessioni, permutazioni e rietichettature è 5.472.730.538.

Storicamente questo gioco è un caso ben più facile da risolvere di un antico e famoso gioco di logica-matematica a cui si è dedicato anche Eulero da Basilea; si tratta dei *quadrati greco-latini*. In questo caso, a differenza del Sudoku, non vi sono griglie interne e l'unica condizione da rispettare è che in ogni riga ed in ogni colonna compaiano tutti i numeri da 1 ad $n \times n$ una volta ed una volta sola, dove $n$ è la dimensione del quadrato (nel caso del Sudoku $n=9$). Inoltre occorre sovrapporre n soluzioni di questo tipo (dette quadrati latini) in modo che ciascuna casella abbia una n-upla distinta.

Al contrario di quanto spesso si afferma, il sudoku è un gioco di logica e non di matematica e non ha a che fare con i numeri. Le proprietà dei numeri non vengono mai utilizzate e neppure viene mai utilizzato il fatto che siano dei numeri. Per rendersi conto della cosa basta pensare che il gioco sarebbe esattamente identico se anziché i primi nove numeri si usassero le prime nove lettere dell'alfabeto oppure nove simboli diversi tra loro (non c'è nemmeno bisogno che tra i simboli sussista un ordine).

Tuttavia alcuni ricercatori matematici hanno messo in evidenza molti legami tra Sudoku e quadrati magici."

(Wikipedia)

**Una definizione rigorosa.**

Il problema del sudoku può essere risolto, cioè è possibile generare schemi validi sudoku, avvalendosi della programmazione lineare (LP).

Un possibile modello per rappresentare il sudoku è il seguente:

Variabili;
Sia, x(i,j,k): variabile binaria che vale 1 se la casella (i,j) ha valore k (1<=k<=9, k intero) e che vale 0 se la casella (i,j) ha valore diverso da k.

Funzione obiettivo;
non c'è funzione obiettivo da ottimizzare dato che si cerca esclusivamente una soluzione ammissibile.

Vincoli;

$$\sum_{k=1}^{9} x(i,j,k) = 1 \quad \forall i,j \quad (1)$$

(ogni elemento della matrice deve contenere un numero da 1 a 9)

$$\sum_{j=1}^{9} x(i,j,k) = 1 \quad \forall i,k \quad (2)$$

(in ogni riga della tabella devono essere presenti tutti i numeri da 1 a 9)

$$\sum_{i=1}^{9} x(i,j,k) = 1 \quad \forall j,k \quad (3)$$

(in ogni colonna della tabella devono essere presenti tutti i numeri da 1 a 9)

$$\sum_{i,j:(i,j)\in RQ-h} x(i,j,k) = 1 \forall h = 1,\dots,9 \forall k = 1,\dots,9 \ (4)$$

(in ogni riquadro della tabella devono essere presenti tutti i numeri da 1 a 9)

Dove RQ-h è l'acchesimo riquadro della tabella, se consideriamo l'intera tabella suddivisa in riquadri da 3x3, otterremo 9 riquadri contenenti ciascuno 9 numeri, ognuno di questi riquadri deve contenere tutti e soli i numeri da 1 a 9.

x(i,j,k) = 1 ∀ elemento (i,j) della matrice iniziale contenente il numero k.

**Ingredienti dell'applicazione.**

L'applicazione è interamente scritta in Java ed è stato utilizzato l'ambiente di sviluppo Eclipse Oxigen con sistema operativo Windows 10 Home.
Per risolvere il modello matematico di programmazione lineare di cui al paragrafo precedente è stato utilizzato il metodo del simplesso. Mi sono avvalso della libreria realizzata dal Dott. Stefano Scarioli denominata SSC.

"SSC (Software per il Calcolo del Simplesso) è una libreria java per la risoluzione di problemi di programmazione lineare. SSC utilizza l'algoritmo del Simplesso per risolvere questa classe di problemi di ottimizzazione di norma indicati con la sigla LP (Linear Programming).

In SSC è possibile risolvere anche problemi con variabili libere, intere, binarie, ***semicontinue e semicontinue intere***. Questa sottoclasse di problemi sono di norma denominati con la sigla MILP (Mixed Integer Linear Programming). Nel caso di problemi MILP, che presentano tutte o una parte di variabili intere o binarie o semicontinue, SSC utilizza l'algoritmo del Branch and Bound (B&B) per la loro risoluzione.

SSC è un progetto open source sotto licenza MIT."

(http://www.ssclab.org)

La libreria SSC prevede quattro formati di input per i problemi; a coefficienti, matriciale, sparso, a disequazioni. Ho scelto, per motivi di praticità, di utilizzare il modello a disequazioni, che è il meno efficiente di tutti, ma è molto comodo dal punto di vista della elaborazione formale da programma. Peraltro in un

breve colloquio avuto con il Dott. Stefano Scarioli, mi ha detto che il modello a disequazioni è l'ultimo che è stato implementato e può sicuramente essere migliorato.

Vediamo ora come ho implementato i quattro tipi di vincolo:

```
Vincolo (1);
String s="",piu="";
  for(int i=0;i<9;i++)
    for(int j=0;j<9;j++) {
      s="";
      piu="";
      for(int k=1;k<10;k++) {
      s=s+piu+"x"+Integer.toString(i)+Integer.toString(j)+
           Integer.toString(k)+" ";
           piu="+"}
       constraints.add(s+" = 1");
      }
```

E' chiaro che ho dovuto adottare una convenzione per la nomenclatura delle variabili, ho utilizzato x seguito da tre cifre da 0 a 8 per le coordinate dello schema, da 1 a 9 per i numeri. Così ad esempio la variabile x777 corrisponde all'elemento x(7,7,7), cioè riga 7, colonna 7, numero 7. In questo modo si è reso necessario utilizzare complessivamente 889 variabili. La variabile x6 è da considerare come x006, e cioè riga 0, colonna 0, numero 6.
Analogamente per i vincoli (2) e (3), semplicemente cambiando l'ordine dei cicli for delle variabili.
Per quanto riguarda il vincolo (4) ho dovuto procedere manualmente, considerando una per una tutte le variabili

dei vari riquadri, vi fornisco l'esempio per i primi 3 riquadri:

```
constraints.add("x001 +x011 +x021 +x101 +x111 +x121 +x201 +x211+ x221 = 1");
	constraints.add("x031 +x041 +x051 +x131 +x141 +x151 +x231 +x241+ x251 = 1");
	constraints.add("x061 +x071 +x081 +x161 +x171 +x181 +x261 +x271+ x281 = 1");
	constraints.add("x301 +x311 +x321 +x401 +x411 +x421 +x501 +x511+ x521 = 1");
	constraints.add("x331 +x341 +x351 +x431 +x441 +x451 +x531 +x541+ x551 = 1");
	constraints.add("x361 +x371 +x381 +x461 +x471 +x481 +x561 +x571+ x581 = 1");
	constraints.add("x601 +x611 +x621 +x701 +x711 +x721 +x801 +x811+ x821 = 1");
	constraints.add("x631 +x641 +x651 +x731 +x741 +x751 +x831 +x841+ x851 = 1");
	constraints.add("x661 +x671 +x681 +x761 +x771 +x781 +x861 +x871+ x881 = 1");
```

La funzione obiettivo, non necessaria in questo modello in quanto è sufficiente una soluzione ammissibile, è stata scelta in modo arbitrario in quanto questa libreria non prevede l'assenza di una funzione obiettivo. Quindi è stata scelto il minimo della somma di tutte le 889 variabili che, se la soluzione è ammissibile, deve risultare pari a 81.
Ho aggiunto un vincolo ulteriore per far si che ad ogni esecuzione la soluzione generata fosse diversa dalla precedente in modo casuale, questo l'ho ottenuto assegnando la prima riga della tabella con tutti e soli i numeri da 1 a 9 generati in posizioni casuali. Si possono adottare anche altre soluzioni, ma all'aumentare delle variabili assegnate inizialmente, aumenta notevolmente il tempo di esecuzione per la risoluzione del simplesso.
Per far si che fosse generato uno schema diverso ad ogni esecuzione dell'applicazione ho aggiunto un ulteriore vincolo; ho generato in sequenza casuale tutti i numeri da 1 a 9 assegnandoli alla prima riga dello schema. Siccome le permutazioni di 9 numeri sono pari a 9!=362.880, questo è anche il numero di schemi diversi che possono essere generati da questo modello. Volendo assegnare, sempre nel rispetto dei vincoli (1),(2),(3),(4), una riga ulteriore di numeri da uno a nove, otterremmo 9!*9!= 131.681.894.400 possibili schemi diversi.

Di seguito il codice che genera il vincolo descritto;

```
int n[],x,flag;

n=new int[10];
x=0;

Random random = new Random();

for(int i=0;i<9;i++)
{
   flag=1;
            n[i]=0;
   while(flag==1)
   {
            flag=0;
            x=(int)(random.nextInt(9)+1);
            for(int j=0;j<i;j++)
                     if (n[j]==x)
                              flag =1;
   }
   n[i]=x;
}

s="";piu="";
for(int i=0;i<9;i++)
{

   s=s+piu+"x"+Integer.toString(0)+Integer.toString(i)+Integ
er.toString(n[i])+" ";
            piu="+";
}
   constraints.add(s+" = 9");
```

Di seguito fornisco il listato completo dell'applicazione Java Sudoku:

```
package com.andreabianchini.sudoku;

import java.util.ArrayList;
import java.util.Random;

import it.ssc.log.SscLogger;
import it.ssc.pl.milp.GoalType;
import it.ssc.pl.milp.LP;
import it.ssc.pl.milp.LinearObjectiveFunction;
import it.ssc.pl.milp.Solution;
import it.ssc.pl.milp.SolutionType;
import it.ssc.pl.milp.Variable;

public class Sudoku {

	static int schema[][];

	public static void main(String[] args) throws Exception {
		// TODO Auto-generated method stub

		schema=new int[9][9];
		generaSchemaFinale();
	}

	public static boolean generaSchemaFinale() throws
Exception
	{

      double c[];
      c = new double[889];

      for(int i=0;i<889;i++)
         c[i]=1;
```

```
        LinearObjectiveFunction fo = new
LinearObjectiveFunction(c, GoalType.MIN);
        ArrayList< String > constraints = new ArrayList< String >();


        String s="",piu="";
        for(int i=0;i<9;i++)
           for(int j=0;j<9;j++)
           {
                    s="";
                    piu="";
              for(int k=1;k<10;k++)
                            {

           s=s+piu+"x"+Integer.toString(i)+Integer.toString(j)+Integ
er.toString(k)+" ";
                                    piu="+";
                            }
                    constraints.add(s+" = 1");
           }

        for(int i=0;i<9;i++)
           for(int k=1;k<10;k++)
           {
                    s="";
                    piu="";
              for(int j=0;j<9;j++)
                            {

           s=s+piu+"x"+Integer.toString(i)+Integer.toString(j)+Integ
er.toString(k)+" ";
                                    piu="+";
                            }
                    constraints.add(s+" = 1");
           }
```

```
    for(int k=1;k<10;k++)
       for(int j=0;j<9;j++)
       {
                s="";
                piu="";
                for(int i=0;i<9;i++)
                      {

        s=s+piu+"x"+Integer.toString(i)+Integer.toString(j)+Integ
er.toString(k)+" ";
                                    piu="+";
                      }
                constraints.add(s+" = 1");
       }

     constraints.add("x001  +x011  +x021  +x101  +x111  +x121
+x201 +x211+ x221 = 1");
     constraints.add("x031  +x041  +x051  +x131  +x141  +x151
+x231 +x241+ x251 = 1");
     constraints.add("x061  +x071  +x081  +x161  +x171  +x181
+x261 +x271+ x281 = 1");
     constraints.add("x301  +x311  +x321  +x401  +x411  +x421
+x501 +x511+ x521 = 1");
     constraints.add("x331  +x341  +x351  +x431  +x441  +x451
+x531 +x541+ x551 = 1");
     constraints.add("x361  +x371  +x381  +x461  +x471  +x481
+x561 +x571+ x581 = 1");
     constraints.add("x601  +x611  +x621  +x701  +x711  +x721
+x801 +x811+ x821 = 1");
     constraints.add("x631  +x641  +x651  +x731  +x741  +x751
+x831 +x841+ x851 = 1");
     constraints.add("x661  +x671  +x681  +x761  +x771  +x781
+x861 +x871+ x881 = 1");

     constraints.add("x002  +x012  +x022  +x102  +x112  +x122
```

```
+x202 +x212+ x222 = 1");
      constraints.add("x032  +x042  +x052  +x132  +x142  +x152
+x232 +x242+ x252 = 1");
      constraints.add("x062  +x072  +x082  +x162  +x172  +x182
+x262 +x272+ x282 = 1");
      constraints.add("x302  +x312  +x322  +x402  +x412  +x422
+x502 +x512+ x522 = 1");
      constraints.add("x332  +x342  +x352  +x432  +x442  +x452
+x532 +x542+ x552 = 1");
      constraints.add("x362  +x372  +x382  +x462  +x472  +x482
+x562 +x572+ x582 = 1");
      constraints.add("x602  +x612  +x622  +x702  +x712  +x722
+x802 +x812+ x822 = 1");
      constraints.add("x632  +x642  +x652  +x732  +x742  +x752
+x832 +x842+ x852 = 1");
      constraints.add("x662  +x672  +x682  +x762  +x772  +x782
+x862 +x872+ x882 = 1");

      constraints.add("x003  +x013  +x023  +x103  +x113  +x123
+x203 +x213+ x223 = 1");
      constraints.add("x033  +x043  +x053  +x133  +x143  +x153
+x233 +x243+ x253 = 1");
      constraints.add("x063  +x073  +x083  +x163  +x173  +x183
+x263 +x273+ x283 = 1");
      constraints.add("x303  +x313  +x323  +x403  +x413  +x423
+x503 +x513+ x523 = 1");
      constraints.add("x333  +x343  +x353  +x433  +x443  +x453
+x533 +x543+ x553 = 1");
      constraints.add("x363  +x373  +x383  +x463  +x473  +x483
+x563 +x573+ x583 = 1");
      constraints.add("x603  +x613  +x623  +x703  +x713  +x723
+x803 +x813+ x823 = 1");
      constraints.add("x633  +x643  +x653  +x733  +x743  +x753
+x833 +x843+ x853 = 1");
      constraints.add("x663  +x673  +x683  +x763  +x773  +x783
+x863 +x873+ x883 = 1");
```

```
        constraints.add("x004 +x014 +x024 +x104 +x114 +x124
+x204 +x214+ x224 = 1");
        constraints.add("x034 +x044 +x054 +x134 +x144 +x154
+x234 +x244+ x254 = 1");
        constraints.add("x064 +x074 +x084 +x164 +x174 +x184
+x264 +x274+ x284 = 1");
        constraints.add("x304 +x314 +x324 +x404 +x414 +x424
+x504 +x514+ x524 = 1");
        constraints.add("x334 +x344 +x354 +x434 +x444 +x454
+x534 +x544+ x554 = 1");
        constraints.add("x364 +x374 +x384 +x464 +x474 +x484
+x564 +x574+ x584 = 1");
        constraints.add("x604 +x614 +x624 +x704 +x714 +x724
+x804 +x814+ x824 = 1");
        constraints.add("x634 +x644 +x654 +x734 +x744 +x754
+x834 +x844+ x854 = 1");
        constraints.add("x664 +x674 +x684 +x764 +x774 +x784
+x864 +x874+ x884 = 1");

        constraints.add("x005 +x015 +x025 +x105 +x115 +x125
+x205 +x215+ x225 = 1");
        constraints.add("x035 +x045 +x055 +x135 +x145 +x155
+x235 +x245+ x255 = 1");
        constraints.add("x065 +x075 +x085 +x165 +x175 +x185
+x265 +x275+ x285 = 1");
        constraints.add("x305 +x315 +x325 +x405 +x415 +x425
+x505 +x515+ x525 = 1");
        constraints.add("x335 +x345 +x355 +x435 +x445 +x455
+x535 +x545+ x555 = 1");
        constraints.add("x365 +x375 +x385 +x465 +x475 +x485
+x565 +x575+ x585 = 1");
        constraints.add("x605 +x615 +x625 +x705 +x715 +x725
+x805 +x815+ x825 = 1");
        constraints.add("x635 +x645 +x655 +x735 +x745 +x755
+x835 +x845+ x855 = 1");
```

```
	constraints.add("x665 +x675 +x685 +x765 +x775 +x785
+x865 +x875+ x885 = 1");

	constraints.add("x006 +x016 +x026 +x106 +x116 +x126
+x206 +x216+ x226 = 1");
	constraints.add("x036 +x046 +x056 +x136 +x146 +x156
+x236 +x246+ x256 = 1");
	constraints.add("x066 +x076 +x086 +x166 +x176 +x186
+x266 +x276+ x286 = 1");
	constraints.add("x306 +x316 +x326 +x406 +x416 +x426
+x506 +x516+ x526 = 1");
	constraints.add("x336 +x346 +x356 +x436 +x446 +x456
+x536 +x546+ x556 = 1");
	constraints.add("x366 +x376 +x386 +x466 +x476 +x486
+x566 +x576+ x586 = 1");
	constraints.add("x606 +x616 +x626 +x706 +x716 +x726
+x806 +x816+ x826 = 1");
	constraints.add("x636 +x646 +x656 +x736 +x746 +x756
+x836 +x846+ x856 = 1");
	constraints.add("x666 +x676 +x686 +x766 +x776 +x786
+x866 +x876+ x886 = 1");

	constraints.add("x007 +x017 +x027 +x107 +x117 +x127
+x207 +x217+ x227 = 1");
	constraints.add("x037 +x047 +x057 +x137 +x147 +x157
+x237 +x247+ x257 = 1");
	constraints.add("x067 +x077 +x087 +x167 +x177 +x187
+x267 +x277+ x287 = 1");
	constraints.add("x307 +x317 +x327 +x407 +x417 +x427
+x507 +x517+ x527 = 1");
	constraints.add("x337 +x347 +x357 +x437 +x447 +x457
+x537 +x547+ x557 = 1");
	constraints.add("x367 +x377 +x387 +x467 +x477 +x487
+x567 +x577+ x587 = 1");
	constraints.add("x607 +x617 +x627 +x707 +x717 +x727
+x807 +x817+ x827 = 1");
```

```
        constraints.add("x637 +x647 +x657 +x737 +x747 +x757
+x837 +x847+ x857 = 1");
        constraints.add("x667 +x677 +x687 +x767 +x777 +x787
+x867 +x877+ x887 = 1");

        constraints.add("x008 +x018 +x028 +x108 +x118 +x128
+x208 +x218+ x228 = 1");
        constraints.add("x038 +x048 +x058 +x138 +x148 +x158
+x238 +x248+ x258 = 1");
        constraints.add("x068 +x078 +x088 +x168 +x178 +x188
+x268 +x278+ x288 = 1");
        constraints.add("x308 +x318 +x328 +x408 +x418 +x428
+x508 +x518+ x528 = 1");
        constraints.add("x338 +x348 +x358 +x438 +x448 +x458
+x538 +x548+ x558 = 1");
        constraints.add("x368 +x378 +x388 +x468 +x478 +x488
+x568 +x578+ x588 = 1");
        constraints.add("x608 +x618 +x628 +x708 +x718 +x728
+x808 +x818+ x828 = 1");
        constraints.add("x638 +x648 +x658 +x738 +x748 +x758
+x838 +x848+ x858 = 1");
        constraints.add("x668 +x678 +x688 +x768 +x778 +x788
+x868 +x878+ x888 = 1");

        constraints.add("x009 +x019 +x029 +x109 +x119 +x129
+x209 +x219+ x229 = 1");
        constraints.add("x039 +x049 +x059 +x139 +x149 +x159
+x239 +x249+ x259 = 1");
        constraints.add("x069 +x079 +x089 +x169 +x179 +x189
+x269 +x279+ x289 = 1");
        constraints.add("x309 +x319 +x329 +x409 +x419 +x429
+x509 +x519+ x529 = 1");
        constraints.add("x339 +x349 +x359 +x439 +x449 +x459
+x539 +x549+ x559 = 1");
        constraints.add("x369 +x379 +x389 +x469 +x479 +x489
+x569 +x579+ x589 = 1");
```

```
        constraints.add("x609  +x619  +x629  +x709  +x719  +x729
+x809 +x819+ x829 = 1");
        constraints.add("x639  +x649  +x659  +x739  +x749  +x759
+x839 +x849+ x859 = 1");
        constraints.add("x669  +x679  +x689  +x769  +x779  +x789
+x869 +x879+ x889 = 1");

        int n[],x,flag;

        n=new int[10];
        x=0;

        Random random = new Random();

        for(int i=0;i<9;i++)
        {
           flag=1;
                   n[i]=0;
           while(flag==1)
           {
                   flag=0;
                   x=(int)(random.nextInt(9)+1);
                   for(int j=0;j<i;j++)
                           if (n[j]==x)
                                   flag =1;
           }
           n[i]=x;
        }

        s="";piu="";
        for(int i=0;i<9;i++)
        {

           s=s+piu+"x"+Integer.toString(0)+Integer.toString(i)+Integ
er.toString(n[i])+" ";
                   piu="+";
```

```
        }
           constraints.add(s+" = 9");


        LP lp = new LP(constraints,fo);
        SolutionType solution_type=lp.resolve();

        if(solution_type==SolutionType.OPTIMUM) {
           Solution soluzione=lp.getSolution();
           for(Variable var:soluzione.getVariables()) {
                    int row,col,num;
                    double val;
                    String st;
                    st=var.getName();
                    row=Integer.parseInt(st.substring(1));
                    st=String.format("%03d", row);
                    row=Integer.parseInt(st.substring(0,1));
                    col=Integer.parseInt(st.substring(1,2));
                    num=Integer.parseInt(st.substring(2,3));
                    val=var.getValue();
                    if (val>0.5)
                             schema[row][col]=num;

              SscLogger.log("Nome variabile :"+var.getName() + "
valore :"+var.getValue());
           }
           SscLogger.log("Valore
ottimo:"+soluzione.getOptimumValue());
        }



                    for(int i=0;i<9;i++)
                    {
                             for(int j=0;j<9;j++)
```

```
                {
                        System.out.print(schema[i][j]);
                }
                System.out.println();
        }

        return false;
    }
}
```

## Programmazione lineare

Da Wikipedia, l'enciclopedia libera.

La **programmazione lineare** (PL) è quella branca della ricerca operativa che si occupa di studiare algoritmi di risoluzione per *problemi di ottimizzazione lineari*.

Un problema è detto *lineare* se sia la *funzione obiettivo* sia i *vincoli* sono funzioni lineari.

Questo significa che la funzione obiettivo può essere

scritta come: avendo indicato con

- *NV* il numero delle variabili che descrivono il problema;
- il vettore colonna dei coefficienti della funzione obiettivo;
- il vettore colonna delle variabili .
- la T ad esponente è l'operatore di trasposizione.

Esistono tre grandi classi di problemi lineari:

1) Problemi lineari continui (**L**inear **P**rogramming =>**LP**)

2) Problemi lineari interi (**I**nteger **L**inear **P**rogramming =>**ILP**)

3) Problemi lineari misto-interi
(**M**ixed **I**nteger **L**inear **P**rogramming => **MILP**)

Un problema di programmazione lineare consiste nel massimizzare o minimizzare una funzione lineare definita sull'insieme delle soluzioni di un sistema di disequazioni lineari, dette *vincoli*.

Per esempio il problema:

max x+2y
y-x <=4
x,y>=0

è un problema di programmazione lineare.

I vincoli definiscono la regione ammissibile (cioè l'insieme dei punti che soddisfano tutti i vincoli del problema, in inglese "feasible region"). Nel caso della programmazione lineare la regione ammissibile è un poliedro che può essere vuoto, limitato o illimitato. La funzione che va minimizzata o massimizzata è la *funzione obiettivo*: essa in pratica calcola il "costo" di ogni soluzione dei vincoli.

È quindi obiettivo della risoluzione di un tale problema trovare tra le soluzioni che soddisfino i vincoli, quella cui corrisponde il costo minimo (o massimo, se si tratta di un ricavo).

# Algoritmo del simplesso

Da Wikipedia, l'enciclopedia libera.

L'**algoritmo del simplesso**, ideato dall'americano George Dantzig nel 1947, è un metodo numerico per risolvere problemi di programmazione lineare. È citato dalla rivista statunitense *Computing in Science and Engineering* come uno dei dieci migliori algoritmi del secolo.[1]

Questo algoritmo fa uso del concetto di simplesso, cioè un politopo di $N+1$ vertici in $N$ dimensioni: un segmento di retta in una dimensione, un triangolo in due dimensioni, un tetraedro in tre dimensioni.