

A Nimatron

Martin J.Chlond
 Lancashire Business School
 University of Central Lancashire
 Preston PR1 2HE
 UK

mchlond@uclan.ac.uk

Oguz Akyol

akyol@ualberta.ca

Editor’s note: This is a pdf copy of an html document which resides at

<http://ite.pubs.informs.org/Vol3No3/ChlondAkyol/>

Nim is a two-person game of perfect information thought to be Chinese in origin. The game begins with a number of heaps of objects, usually coins or matchsticks. The players alternate moves by taking any number of objects from any single pile. The player making the last move is the winner. A Java applet to play the game is available on the INFORMS website <http://ite.pubs.informs.org/Vol3No3/ChlondAkyol/Nim.php>.

The winning strategy for Nim is attributed to Bouton (1902) and consists of expressing the heap sizes in binary notation and summing the digits of the same power of 2, modulo 2. A winning position is one where all digits of the reduced sum are zero. Each player therefore will attempt to remove objects from heaps in order to achieve such a position. Once a winning position is achieved by one player then the opposing player has no option but to adopt a losing position, that is, he is forced to move such that at least one binary column has a reduced sum of 1.

The solution identified by Bouton is easily implemented and in fact Martin Gardner (1959) mentions a number of mechanical devices constructed in the nineteen-forties and nineteen-fifties that played a perfect game of Nim. Most notable of these was the Nimatron, built in 1940 and weighing a ton. Nevertheless, the identification of an appropriate winning move may be formulated as an IP and provides a neat student exercise.

FORMULATION

n_i = number of objects in heap i before move

h = number of heaps

c = number of columns required for binary conversion

Define variables as follows:

s_i = number of objects taken from heap i

m_i = number of objects in heap i after move

$x_{i,j}$ = binary representations of heap sizes after move

d_i = 1 if heap i changed, 0 otherwise

w_i = dummy variables for winning position test

Minimize the number of heaps changed.

$$\text{Minimize } \sum_{i=1}^h d_i$$

If the solution is zero then no winning move is available for the current position.

Convert heap numbers (after move) to binary.

$$\sum_{j=1}^c 2^{j-1} x_{i,j} = m_i \quad \forall i$$

Ensure safe position after move.

$$\sum_{i=1}^h x_{i,j} = 2w_j \quad \forall j$$

Ensure that heap sizes before and after the move are consistent with the move itself.

$$n_i - s_i = m_i \quad \forall i$$

Dummy variables for each heap are set to 1 if heap is changed.

$$s_i - M d_i \leq 0 \quad \forall i$$

M is an upper bound on heap size.

Download Xpress-Mosel code to implement the model here.

MOORE'S GAME

A modification of Nim proposed by E.H.Moore (1910) is where players may remove objects from up to k heaps in a single move.

As with standard Nim, heap sizes are expressed in binary notation and the digits of the same power of two are summed. The sums are reduced modulo k+1 and a winning position is where all digits of the reduced sum are zero.

Hence, the formulation to identify a winning move from a given position is identical to that for standard Nim except a winning position is ensured by:

$$\sum_{i=1}^h x_{i,j} = (k+1)w_j \quad \forall j$$

Nim therefore is Moore's Game where $k = 1$.

The Xpress-Mosel model above may be used to identify winning positions for Moore's game by changing the value of k.

ROSEBUSHES

A further modification is where a single object may be removed from up to k heaps. This was proposed by Schuh (1968), embellished and investigated by de Carteblanche (1970,1974) and given further consideration by Berlekamp, Conway and Guy (1982) and Vajda (1992).

The game has not been fully investigated although solutions to a number of starting configurations are known. In particular, Vajda (1992) describes the progress of a game with 5 heaps and $k = 2$.

Arrange the heap sizes in ascending order. Winning positions are odd-odd-odd-even-even, odd-even-even-odd-odd, even-even-odd-odd-odd and even-even-even-even-even.

An Xpress-Mosel model to identify winning positions for this configuration is here.

REFERENCES

- Berlekamp, E.R., Conway, R.H. and R.K. Guy (1982), *Winning Ways for your mathematical plays - Volume 2: Games in particular*, Academic Press.
- Bouton, C. L. (1902), "Nim, A Game with A Complete Mathematical Theory," *Annals of Math*, Vol. 3, pp. 35-39.
- de Carteblanche, F. (1970), "The Princess and the Roses," *J. Recr. Math.*, Vol. 3, pp. 238-239.
- de Carteblanche, F. (1974), "The Roses and the Princess," *J. Recr. Math.*, Vol. 7, pp. 295-298.
- Gardner, M. (1959), *Mathematical Puzzles and Diversions*, Penguin
- Moore, E. H. (1910), "A Generalization of the Game Called Nim," *Ann. of Math.*, Vol. 11, pp. 93-94.
- Schuh, F. (1968), *The Master Book of Mathematical Recreations*, Dover.
- Vajda, S. (1992), *Mathematical Games and How to Play Them*, Ellis Horwood

APPENDICES

Identify winning move in Nim - Xpress-Mosel model

model 'nim'

! nim.mos : Computes move to safe position (if available) in game of Nim
! Written by : Martin J. Chlond
! Date written : 26 December 2001

uses 'mmxprs'
parameters

heap = 5 ! number of heaps
col = 4 ! columns for binary representation of position after move
k = 1 ! maximum number of heaps to change (if k>1 then Moore's game) ! number of squares per side
end-parameters

declarations
nmax=2^{col}-1 ! maximum number allowed in any heap
n: array(1..heap) of real
!variables
x: array(1..heap,1..col) of mpvar ! binary values of position after move
d: array(1..heap) of mpvar ! 1 if heap changed, 0 otherwise
s: array(1..heap) of mpvar ! number taken from heap
m: array(1..heap) of mpvar ! number in each heap after move
w: array(1..col) of mpvar ! dummy variables for winning position test
end-declarations

n:=[5,4,3,2,1] ! number in each heap before move

! number of heaps changed
heapch:= sum(i in 1..heap) d(i)

! convert heap numbers (after move) to binary
forall(i in 1..heap)
conb(i):= sum(j in 1..col) 2^(j-1)*x(i,j) = m(i)

! ensures safe position after move
forall(j in 1..col)
winp(j):= sum(i in 1..heap) x(i,j) = (k+1)*w(j)

! positions before and after are consistent with move
forall(i in 1..heap)
cons(i):= n(i)-s(i) = m(i)

! dummy set to 1 if heap changed
forall(i in 1..heap)
dset(i):= s(i)-nmax*d(i) <= 0

```
forall(i in 1..heap,j in 1..col)
x(i,j) is_binary
forall (i in 1..heap) do
d(i) is_binary
s(i) is_integer
m(i) is_integer
end-do
forall(j in 1..col)
w(j) is_integer
```

```
! minimise number of heaps changed - if solution is zero then current position already safe
minimise( heapch )
```

```
! output solution
forall(i in 1..heap) do
write(getsol(s(i)), ' ', getsol(m(i)))
writeln
end-do
```

```
end-model
```

Identify Winning move in Rosebushes - Xpress-Mosel model

```
model 'rosebush'
```

```
! Rosebush.mos : Computes move to safe position (if available) in game of Rosebushes
! Written by : Martin J. Chlond
! Date written : 6 January 2002
! References : Vajda, S., Mathematical Games and how to play them (pp 59,59)
! Schuh, F., The Master Book of Mathematical Recreations (*109 pp 141,142)
! Berlekamp, E.R., Conway, J.H., Guy, R.K., Winning Ways for your Mathematical Plays
```

```
uses 'mmxprs'
```

```
parameters
heap = 5 ! number of heaps
nmax = 16 ! maximum number allowed in any heap
k = 2 ! maximum number of heaps to change
end-parameters
```

```
declarations
! data tables
n: array(1..heap) of real
z: array(1..4) of real
! variables
d: array(1..heap) of mpvar ! 1 if heap changed, 0 otherwise
m: array(1..heap) of mpvar ! number in each heap after move
e: array(1..heap) of mpvar ! 1 if heap odd after move, 0 if even
```

b: array(1..heap) of mpvar ! dummy variable for parity check of heaps
w: array(1..4,1..3) of mpvar ! dummy variables for safety check
s: mpvar ! decimal equivalent of heap parities
end-declarations

n:=[1,1,2,3,4] ! number in each heap before move (example from Vajda)
z:=[0,7,19,28] ! decimal equivalents of 'safe' heap parities (see Vajda)

! minimise number of heaps changed - if solution is zero then current position already safe
heapch:= sum(i in 1..heap) d(i)

! maximum of k rows to change
maxh:= sum(i in 1..heap) d(i) <= k

! ensures safe position after move
forall(i in 1..4) do
lca(i):= s-(32-z(i))*w(i,1) <= z(i)-1
lcb(i):= s-z(i)*w(i,1) >= 0
lcc(i):= s+(z(i)+1)*w(i,2) >= z(i)+1
lcd(i):= s+31*w(i,2) <= z(i)+31
lce(i):= w(i,3) = w(i,1)+w(i,2)-1
end-do
es:= sum(i in 1..4) w(i,3) = 1

! positions before and after are consistent with move
forall(i in 1..heap)
cons(i):= n(i)-d(i) = m(i)

! e(i) = 1 if m(i) odd, 0 otherwise
forall(i in 1..heap)
eset(i):= m(i) = 2*b(i)+e(i)

! computes decimal equivalent of heap parities
sset:= sum(i in 1..heap) 2^(heap-i)*e(i) = s

! ensures piles remain in increasing order
forall(i in 1..heap-1)
ndec(i):= m(i+1) >= m(i)

forall(i in 1..heap) do
d(i) is_binary
m(i) is_integer
e(i) is_binary
b(i) is_integer
end-do

forall(i in 1..4,j in 1..3)
w(i,j) is_binary

! minimise number of heaps changed - if solution is zero then current position already safe
 minimise(heapch)

```
! output solution
forall(i in 1..heap) do
write(getsol(d(i)), ' ', getsol(m(i)))
writeln
end-do
```

end-model

Source Code of Nim

```
// Nim.java
import java.awt.*;
import java.applet.Applet;

public class Nim extends Applet
{
    // initialise game
    private Image coin;
    private int currentImage = 0;
    int nPiles; // number of piles
    int sPiles[]; // size of piles
    int move[] = new int[3]; // array to store current move
    int numLeft; // total number of items left
    int rowPos, colPos; // row and column positions
    int xDown, yDown; // pointer position at mouse click
    boolean mover; // next mover - true = player
    // false = computer

    private Graphics gContext;
    private Image buffer;

    public void start()
    {
        setBackground(Color.white);
        coin = getImage(getDocumentBase(), "rcoin.gif");
        setLayout(new BorderLayout());
        nPiles = genPiles();
        sPiles = new int[nPiles];
        sPiles = makePile(nPiles);
        numLeft = totStones(nPiles, sPiles);
        mover = true;
        buffer = createImage(650, 370);
        gContext = buffer.getGraphics();
        gContext.setColor(Color.white);
        gContext.drawImage(buffer, 0, 0, this);
    }
}
```

```
public void paint(Graphics g)
{
    dispGame(g,nPiles,sPiles);
    g.drawImage(buffer,0,0,this);
}

public void update(Graphics g)
{
    paint(g);
}

// play game
public void play()
{
    if(mover)
    {
        move = playMove(nPiles,sPiles);
        if(move[2] == 0)
        {
            mover = !mover;
            numLeft -= move[1];
            sPiles[move[0]] -= move[1];
            repaint();
        }
        else
            showStatus("Invalid move");
    }

    if(!mover && numLeft > 0)
    {
        move = compMove(nPiles,sPiles);
        mover = !mover;
        numLeft -= move[1];
        sPiles[move[0]] -= move[1];
        repaint();
    }

    if(numLeft==0)
        showResult(mover);
}

public boolean mouseDown( Event e, int x, int y)
{
    xDown = x;
    yDown = y;
    play();
    return true;
}
```

```
// generate number of piles
int genPiles()
{
    int piles;
    piles = 4+(int)(Math.random()*3);
    return piles;
}

// make piles
int [] makePile(int nPiles)
{
    int size[] = new int[nPiles];

    for(int i=0;i < nPiles;i++)
        size[i] = 1+(int)(Math.random()*9);

    return size;
}

// total stones
int totStones(int nPiles,int sPiles[])
{
    int Stones = 0;

    for(int i=0;i < nPiles;i++)
        Stones += sPiles[i];

    return Stones;
}

// display game
public void dispGame(Graphics g, int nP, int sP[])
{
    gContext.fillRect(0,0,650,370);
    gContext.setColor(Color.black);
    gContext.drawRect(0,0,649,369);
    gContext.setColor(Color.white);
    for(int i = 0; i < nP; i++)
        printRow(g,sP[i],i);
}

// print row
void printRow(Graphics g, int psize, int i)
{
    int colPos,rowPos;
    rowPos = 5+i*60;
    for(int j = 1; j <= psize; j++)
    {
```

```

        colPos = 30+j*60;
        gContext.drawImage(coin,colPos,rowPos,this);
    }
}

// Player's move
int [] playMove(int nP, int sP[])
{
    int pmove[] = new int[3];

    pmove[0] = (int) ((yDown-5)/60);
    pmove[1] = sP[pmove[0]]-(int) ((xDown-30)/60)+1;
    pmove[2] = 0;

    if(pmove[1] > sP[pmove[0]] | pmove[1] <= 0)
        pmove[2] = 1;

    showStatus("Player takes "+Integer.toString(pmove[1])
              +" items from row "+Integer.toString(pmove[0]+1));
    return pmove;
}

// Computer's move
public int [] compMove(int nP, int sizeP[])
{
    int cmove[] = new int[3];
    int binary[][] = new int[6][4];
    int power[] = new int[4];
    int t[] = new int[4];
    int f = 0;
    int sP;

    power[0]=1;power[1]=2;power[2]=4;power[3]=8;

    // convert pile sizes to binary
    for(int i=0;i<nP;i++)
    {
        sP = sizeP[i];
        for(int j=3;j>=0;j--)
        {
            if(sP<power[j])
                binary[i][j]=0;
            else
            {
                binary[i][j]=1;
                sP -= power[j];
            }
        }
    }
}

```

```
for (int i=0;i<4;i++)
    for(int j=0;j<nP;j++)
        t[i] = t[i] + binary[j][i];

for (int i=0;i<4;i++)
    if(2*(t[i]/2) != t[i])
        f = i;

for (int j=0;j<nP;j++)
    if(binary[j][f]==1)
        cmove[0]=j;

for (int i=3;i>=0;i--)
{
    if(binary[cmove[0]][i]==1 && 2*(t[i]/2) != t[i])
        cmove[1] = cmove[1]+power[i];
    if(binary[cmove[0]][i]==0 && 2*(t[i]/2) != t[i])
        cmove[1] = cmove[1]-power[i];
}

int i = 0;
while(cmove[1]==0)
{
    if(sizeP[i]>0)
    {
        cmove[0] = i;
        cmove[1] = 1+(int)(Math.random()*sizeP[i]);
    }
    i++;
}

showStatus("Nimatron takes "+Integer.toString(cmove[1])
           +" items from row "+Integer.toString(cmove[0]+1));

return cmove;
}

// Show game result
void showResult(boolean mover)
{
    if(mover)
        showStatus("Nimatron Wins!");
    else
        showStatus("You Win!");
}
}
```