David Bozarth
CES 512, Spring 2005

Homework 4 : Peg-Solitaire problem

The problem is to determine whether an arbitrary square Peg-Solitaire board configuration can be played to a win. If the board can not win, then report that fact. Otherwise, demonstrate a sequence of winning moves.

I used the Java BigInteger class to represent an 8 by 8 board configuration in a natural manner, with the lowest-order bit representing the upper-left corner position, bit one representing the next position to the right in the topmost row, bit 8 representing the leftmost position in the second row from the top, … , and bit 63 representing the bottom right corner position.

Keyboard input is inverted left-to-right before representing the board internally, and internal representation is inverted left-to-right before display on the screen. This is to maintain consistency, since standard keyboard-input conversion interprets digits to the left side as higher-order than those to the right, and since this application calls for the leftmost digit to represent the low-order bit.

Each jump generates a new board configuration, which may or not be a winning configuration. On facing a new board configuration, the only constructive choices available to the player are: to determine that the board is a winner, or to determine that the board is a loser, or to perform a jump.

The algorithm relies on a value k defined as:

```
      k = min( n / dim, n % dim ) ,

            where n and dim are non-negative integers,
            dim is the board dimension, and 0 <= n < dim*dim.
```

The definition implies that k is a non-negative integer and that 0 <= k < dim.

The definition results in “corner” or “base” k-values at positions

```
      k + k * dim,
```

Thus n=0 is the base for k=0, n=9 is the base for k=1, and so on.

From the “base” value for a given k, the n-values assigned to the k-category can be determined from

```
n = base, base + 1, ..., base + limit, base + dim, base + 2*dim, ..., base + limit*dim,

      where limit =  dim – (k + 1).
```

This plot of the k-number for each position in the board with dim = 8, represents a Young tableau.

```
0   0   0   0   0   0   0   0
0   1   1   1   1   1   1   1
0   1   2   2   2   2   2   2
0   1   2   3   3   3   3   3
0   1   2   3   4   4   4   4
0   1   2   3   4   5   5   5
0   1   2   3   4   5   6   6
0   1   2   3   4   5   6   7
```

The algorithm’s basic plan to search for a new move is to scan the current lowest-k region first, then if that fails to turn up a jump, then scan the next higher-k region. On encountering region 6, no more moves are possible (since a move requires either 3 contiguous horizontal positions, or 3 contiguous vertical positions). At that time the only constructive choice is to scan the entire board, looking for a singular filled position. In this case the board wins; otherwise it loses.

The next board move is determined completely by the existing board, the current k-value, and an index of the linear “progress” within the current k-based search for moves. This ensures that the function call and return mechanism can serve as the path for backtracking.

Intuitively, since each move removes one peg from the board, and since the “flow” of new moves tends from top left to bottom right of the board, we can visualize the sequence of new boards “running out” of possible new moves in the upper left region, and eventually focusing on the lower right region. As the board becomes more sparse, the likelihood of finding a winning configuration increases – as long as a path for jumps remains among the remaining pegs. In other words, for a win the board needs to thin out, but not get too thin too quickly.

A difficulty is that when a move is found and a new board generated, this may be the same board that was previously checked. If so, then checking this board completely will entail repeating all the moves that occurred before as well. This results in exponential running time. To improve performance, a hashtable is employed to “memoize” losing board configurations so that they would not have to be repeatedly checked, thereby generating other configurations to check, etc.

There were optimizations that could have been fashioned to shave constant-time performance, but these would not be significant compared with the performance improvement incurred by using the hashtable.

```
Algorithm:

BOOLEAN win (k, board)

      if k > 5 then
            // no new board move is possible starting from here
            count all the ones in the board and RETURN true or false

      if this board is in the hashtable, then RETURN false

      kk = k * 8 + k    // gives the "corner" position of the new row/column

      for i = 0 to (8 - k)

            if a jump (new board move) exists in the right-horizontal triplet
starting with (kk + i), then

                  k0 = k < 3 ? 0 : k - 2        // need to re-check the band
of 2 k-numbers before this one

                  BOOLEAN flag = win(k0, new_board)

                  if flag == TRUE then RETURN flag,
                        else write new_board to hashtable

            if a new board move exists in the low-vertical triplet starting
with (kk + i), then

                  k0 = k < 3 ? 0 : k - 2

                  BOOLEAN flag = win(k0, new_board)

                  if flag == TRUE then RETURN flag,
                        else write new_board to hashtable

            if a new board move exists in the right-horizontal triplet
starting with (kk + i * 8), then

                  k0 = k < 3 ? 0 : k - 2

                  BOOLEAN flag = win(k0, new_board)

                  if flag == TRUE then RETURN flag,
                        else write new_board to hashtable

            if a new board move exists in the low-vertical triplet starting
with (kk + i * 8), then

                  k0 = k < 3 ? 0 : k - 2

                  BOOLEAN flag = win(k0, new_board)

                  if flag == TRUE then RETURN flag,
                        else write new_board to hashtable
```

Sample output

```
Elapsed time: 0 hrs 0 min 6.438 sec

Final hash table size = 15632
Hashing all failures.

It is possible to win, starting from this board.

A winning series of moves is:

00100100 01010011 01100001 00010001 00100010 00000011 00010110 00010110
01100100 00010011 00100001 00010001 00100010 00000011 00010110 00010110
00010100 00010011 00100001 00010001 00100010 00000011 00010110 00010110
00000100 00000011 00110001 00010001 00100010 00000011 00010110 00010110
00000100 00010011 00100001 00000001 00100010 00000011 00010110 00010110
00000100 00010100 00100001 00000001 00100010 00000011 00010110 00010110
00000000 00010000 00100101 00000001 00100010 00000011 00010110 00010110
00000000 00010000 00100100 00000000 00100011 00000011 00010110 00010110
00000000 00010000 00100100 00000000 00100011 00010011 00000110 00000110
00000000 00010000 00100100 00000010 00100001 00010001 00000110 00000110
00000000 00010000 00100100 00000011 00100000 00010000 00000110 00000110
00000000 00010000 00100100 00000100 00100000 00010000 00000110 00000110
00000000 00010100 00100000 00000000 00100000 00010000 00000110 00000110
00000000 00010100 00100000 00000000 00100000 00010000 00001000 00000110
00000000 00010100 00100000 00000000 00100000 00010000 00001000 00001000
00000000 00010100 00100000 00000000 00100000 00011000 00000000 00000000
00000000 00010100 00100000 00000000 00100000 00100000 00000000 00000000
00000000 00010100 00100000 00100000 00000000 00000000 00000000 00000000
00000000 00110100 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00001100 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00010000 00000000 00000000 00000000 00000000 00000000 00000000
```

Results

I ran the program on two machines:

(1) HP Pavilion tower
AMD 1.0GHz CPU
256K cache, 200 MHz front side bus
512MB SDRAM 100MHz

(2) Toshiba Satellite notebook
Pentium4 1.6GHz CPU
2MB L2 cache, 400 MHz front side bus
512MB DDR RAM

The running times for the default starting pattern were:

Machine	hash all failures	hash failures at depth 15	no hashing
Pavilion	6.4 s	22 m	36 m
Satellite	2.2 s	8.5 m	16 m