University of the Philippines - Diliman
Department of Computer Science

CS 21 - Machine Problem 1

Peg Solitaire Solver

Author:
Andrei L. Tiangco
altiangco@up.edu.ph

November 08, 2022

Part 1

Introduction

The Peg Solitaire solver, given a 7×7 board of pegs & holes and inaccessible portions:

- checks if a valid solution exists where the final peg is at a certain position, and
- outputs the series of paths that led to the solved board

which is the full implementation or **Implementation C** as specified in the machine problem. We define a solution to be *valid* if the resulting final configuration of the board contains only a single peg and is at its correct final position.

1.1 Input/Output

The input to the program contains 7 lines which make up the rows of the board. Each line consists of 7 adjacent characters–either a peg 'o', a hole '.', an inaccessible portion 'x', a destination hole 'E', or an occupied destination hole 'O'. As per problem specifications, the board should always contain a single destination hole (empty or occupied) and a maximum of 8 pegs.

The output is a line with the string `YES` if a valid solution is found and `NO` if otherwise. Further, if a valid solution is found, paths (sequence of jumps) that led to the valid solution are displayed in succeeding lines.

1.1.1 Sample Input

Assume that the file `input.in` consists of the following:

```
xx...xx
xx...xx
.......
..ooo..
..ooE..
xx...xx
xx...xx
```

1.1.2 Sample Output

Using the command `java -jar mars4_5.jar cs21project1C.asm < input.in`, a corresponding output to the sample input is

```
YES
5,3->3,3
4,5->4,3
3,3->5,3
5,3->5,5
```

1.2 Approach

A naive description of the main program is as follows:

a. Get the input 7×7 board

b. Find and store coordinates of final destination ('O' or 'E')

c. If destination is initially occupied, mutate its content to a peg

d. Else if destination is initially empty, mutate its content to a hole

e. Count and store the total number of pegs in the board

f. Invoke the solver function

The solver function itself employs **recursive backtracking** in the sense that it attempts to reach a valid final configuration using each element in the board as a potential source for each possible jump, recursively calling the solver algorithm until a solution is found or until no further jumps can be made. In the latter case, it backtracks to the pre-jump configuration and proceeds to the next element as another potential source for the jumps.

We analyze the pseudocode of the Peg Solitaire solver as provided in the next page. Observe that there are four requisites to the algorithm, namely the board array, path array, number of pegs, and the coordinates of the final destination given by (x, y).

The base case tells us that a valid solution is found if there is only a single peg left in the board and it is situated in the correct (x, y) location. Otherwise, we iterate through each element using the row and column counters and perform an 'early check' if any jumps can be made by verifying if the current element is a peg. If it is indeed a peg, then we may iterate through each jump direction denoted by $dir \in [0, 4]$, where $(0, 1, 2, 3) := (top, left, bottom, right)$, and perform any valid jump from the current (source) element. The path, which is a string of length 8 and is denoted by `r,c->r',c'` is then pushed to the path array. Using the new configuration of the board, we now recursively call the solver function and ultimately return 1 if the sequence of calls (jumps) lead to a valid final configuration. Otherwise, we perform backtracking by undoing the current jump performed and popping its equivalent path string from the path array.

If all sequence of possible jumps (row, col, dir) have been exhausted yet no valid solution is still found, the algorithm will return 0.

Algorithm Solve Peg Solitaire

Require: $board[7][7], path[8][8], pegs \in [0, 8], x, y \in [0, 7)$
    function PEGSOLVE
        if $pegs = 1$ **and** $board[x][y] =$ 'o' **then**
            return 1
        else
            $row \leftarrow 0$
            while $row < 7$ **do**
                $col \leftarrow 0$
                while $col < 7$ **do**
                    $dir \leftarrow 0$
                    if $board[row][col] =$ 'o' **then**
                        while $dir < 4$ **do**
                            if ISVALIDJUMP(*row,col,dir*) **then**
                                MAKEJUMP(*row,col, dir*)
                                PUSHPATH()
                                if PEGSOLVE() **then**
                                    return 1
                                else
                                    UNDOJUMP(*row,col, dir*)
                                    POPPATH()
            return 0

Part 2

Source Code

The Peg Solitaire solver was initially derived from a pseudocode description before being roughly implemented in C code and finally being translated to MIPS code. With it comes primarily five auxiliary functions, namely *find destination, count pegs, is valid jump, make jump,* and *undo jump*. We elaborate the first four auxiliary functions along with the main solver function and provide relevant code snippets in the succeeding sections.

2.1 High-level Code

2.1.1 Find Destination

The `findDest` function searches for the coordinates of the specific hole the final peg must end up in (i.e. coordinates of 'O' or 'E') and stores each of them in pre-declared variables. The specific hole is then either modified to a peg if it is initially occupied or a hole if it is initially empty. This essentially adjusts Implementation A to B.

```
void findDest(char board[N][N], int *x, int *y) {
  for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
          if (board[i][j] == 'O') {
              *x = i;
              *y = j;
              board[i][j] = 'o';
          }
          else if (board[i][j] == 'E') {
              *x = i;
              *y = j;
              board[i][j] = '.';
          }
      }
  }
}
```

Code Block 2.1: Find Final Destination

2.1.2 Count Pegs

The `countPegs` function basically counts and returns the total number of pegs present in the current configuration of the board.

```
int countPegs(char board[N][N]) {
  int total = 0;
  for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
          if (board[i][j] == 'o') total++;
      }
  }
  return total;
}
```

Code Block 2.2: Count Pegs

2.1.3 Check if Jump is Valid

The `isValidJump` function checks if a jump denoted by (row, col, dir) is valid assuming that the source element contains a peg. A valid jump is defined on a source peg jumping over another peg to an empty hole.

```
int isValidJump(char board[N][N], int row, int col, int dir) {
  // Top jump
  if (dir == 0)
      return (row >= 2 && board[row-1][col] == 'o' && board[row-2][col] == '.');

  // Left jump
  if (dir == 1)
      return (col >= 2 && board[row][col-1] == 'o' && board[row][col-2] == '.');

  // Bottom jump
  if (dir == 2)
      return (row <= 4 && board[row+1][col] == 'o' && board[row+2][col] == '.');

  // Right jump
  else
      return (col <= 4 && board[row][col+1] == 'o' && board[row][col+2] == '.');
}
```

Code Block 2.3: Check if Valid Jump

2.1.4 Make Jump

The `makeJump` function initiates the jump assuming that it is valid, converting the destination element to a peg and the rest of the affected elements into a hole. The reverse of this functionality is implemented via the function `undoJump`.

```
void makeJump(char board[N][N], int row, int col, int dir) {
  board[row][col] = '.';
  if (dir == 0) {
```

```
        board[row-1][col] = '.';
        board[row-2][col] = 'o';
    }
    else if (dir == 1) {
        board[row][col-1] = '.';
        board[row][col-2] = 'o';
    }
    else if (dir == 2) {
        board[row+1][col] = '.';
        board[row+2][col] = 'o';

    }
    else {
        board[row][col+1] = '.';
        board[row][col+2] = 'o';
    }
}
```

Code Block 2.4: Make Jump

2.1.5 Solve Peg Solitaire

The `pegSolve` function is the main solver function to Peg Solitaire. It uses recursive backtracking with the base case being the existence of only a single peg in the board and it being on the correct final location.

```
int pegSolve(char board[N][N], int pegs, int finalX, int finalY) {
  if (pegs == 1 && board[finalX][finalY] == 'o') return 1;
  // For each jump defined by (row, col, dir)
  for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) {
          if (board[i][j] == 'o') {
              for (int k = 0; k < 4; k++) {
                  if (isValidJump(board,i,j,k)) {
                      makeJump(board, i, j, k);
                      // printBoard(board);
                      pegs--;
                      if (pegSolve(board, pegs, finalX, finalY)) return 1;
                      pegs++;
                      undoJump(board,i,j,k);
                      // printBoard(board);
                  }
              }
          }
      }
  }
  return 0;
}
```

Code Block 2.5: Solve Peg Solitaire

2.2 Shift from High-level to Assembly

The problem with our current high-level code is that everything is encapsulated in functions, which leads to a redundancy and abundance of parameter-passing. We patch this going into the assembly code by introducing globals.

2.2.1 Usage of Registers

Temporary Registers

- `$t0`: used for conditionals (e.g. if pegs != 1)
- `$t1`: used solely for array offset (i.e. row*7 + col)
- `$t2`: used for assignment statements (i.e. pegs = pegs - 1)
- `$t3`: start pointer to path array
- `$t4`: final pointer to path array
- `$t5`: used for row/col index to ASCII conversion
- `$t6`: holds row' (current dest row) to be converted to ASCII
- `$t7`: holds col' (current dest col) to be converted to ASCII

Saved Registers

- `$s0`: holds board size/row counter
- `$s1`: holds address of board array
- `$s2`: column counter
- `$s3`: peg counter

Global Pointer

- `0($gp)`: x coordinate of final destination
- `1($gp)`: y coordinate of final destination
- `2($gp)`: total number of pegs

2.2.2 Equivalences

```
.eqv    peg         111              # 'o'
.eqv    hole        46               # '.'
.eqv    board_size  7
```

2.2.3 Data Segment

The 7×7 board is stored as a flattened array in the data segment, and so we may eventually use a single offset to add to the base address in traversing the board array. The path array is also stored as a flattened array, but in this case we make use of a start and final pointer for traversal.

```
.data
board:  .space      49
path:   .space      64
yes:    .asciiz     "YES"
no:     .asciiz     "NO"
```

2.3 Assembly Code

2.3.1 Initialize Board

The `init_board` function is added to consider user input in determining the layout of the Peg Solitaire board.

```
init_board:
  # PREAMBLE GOES HERE
  li      $s0, board_size           # counter for 7 lines
  la      $s1, board                # load start of board address
init_loop:
  beqz    $s0, init_return          # return after exhausting 7 lines
  li      $v0, 8                    # read string
  move    $a0, $s1                  # pass curr address in board
  li      $a1, 8
  syscall
  subi    $s0, $s0, 1               # decrement counter
  addi    $s1, $s1, board_size      # move curr board address by 7 bytes
  j       init_loop
init_return:
  # END GOES HERE
  jr    $ra
```

Code Block 2.6: Initialize Board

2.3.2 Find Destination

`find_dest` now uses the global pointer to store the coordinates of the final destination.

```
find_dest:
  # PREAMBLE GOES HERE
  li    $s0, 0                      # currRow = 0
  la    $s1, board                  # s1 = base
dest_loop:
  beq   $s0, board_size, dest_return  # if currRow == 7, return
  li    $s2, 0                      # currCol = 0
dest_innerloop:
```

```
  beq   $s2, board_size, dest_incrementRow
  lb    $t0, ($s1)                    # access char at current address
  beq   $t0, ’O’, dest_O              # if element == ’O’
  beq   $t0, ’E’, dest_E              # if element == ’E’
  j   dest_incrementCol
dest_O:
  sb    $s0, 0($gp)                   # set global var to final X coord
  sb    $s2, 1($gp)                   # set global var to final Y coord
  assignChar($s1, peg)                # if elem == ’O’, transform to peg
  j   dest_incrementCol
dest_E:
  sb    $s0, 0($gp)                   # set global var to final X coord
  sb    $s2, 1($gp)                   # set global var to final Y coord
  assignChar($s1, hole)               # if elem == ’E’, transform to hole
dest_incrementCol:
  addi    $s2, $s2, 1                 # increment column counter
  addi    $s1, $s1, 1                 # access next elem in flattened array
  j   dest_innerloop                  # iterate through inner loop
dest_incrementRow:
  addi    $s0, $s0, 1                 # increment row counter
  j   dest_loop                       # iterate through outer loop
dest_return:
  # END GOES HERE
  jr    $ra
```

Code Block 2.7: Find Final Destination (MIPS)

2.3.3 Count Pegs

`count_pegs` now uses the global pointer to store the total number of pegs in the board.

```
count_pegs:
  # PREAMBLE GOES HERE
  li      $s3, 0                        # total = 0
  li      $s0, 0                        # currRow = 0
  la      $s1, board
count_loop:
  beq     $s0, board_size, count_return # if currRow == 7, return total
  li      $s2, 0                        # currCol = 0
count_innerloop:
  beq     $s2, board_size, count_incrementRow
  lb      $t0, ($s1)                    # access char/element at current address
  bne     $t0, peg, count_incrementCol  # if element is a peg, total += 1
count_incrementTotal:
  addi    $s3, $s3, 1                   # increment number of pegs
count_incrementCol:
  addi    $s2, $s2, 1                   # increment column counter
  addi    $s1, $s1, 1                   # move to next element in flat array
  j   count_innerloop                   # iterate through inner loop
count_incrementRow:
  addi    $s0, $s0, 1                   # incrememnt row counter
  j       count_loop                    # iterate through outer loop
```

```
count_return:
  sb      $s3, 2($gp)
  # END GOES HERE
  jr    $ra
```

Code Block 2.8: Count Pegs (MIPS)

2.3.4 Check if Jump is Valid

`valid_jump` uses the `get_element` macro (or in my case, pseudo-function) to transform the row, col coordinates into a working memory address in the board array. Their content is then loaded to either check if they are a peg that the source peg can jump over, or if they are a hole that the source peg can land onto.

```
valid_jump:
  # PREAMBLE GOES HERE
  move    $s0, $a0
  move    $s1, $a1
  move    $s2, $a2
  beq   $s2, 1, valid_leftJump      # if dir == 1, check valid left jump
  beq   $s2, 2, valid_bottomJump    # if dir == 2, check valid bottom jump
  beq   $s2, 3, valid_rightJump     # if dir == 3, check valid right jump
valid_topJump:                      # else, check valid top jump
  blt   $s0, 2, valid_return0
  subi    $a0, $s0, 1
  get_element_address($a0, $a1)
  lb    $v0, ($v0)                  # get board[row-1][col]
  bne   $v0, peg, valid_return0
  subi    $a0, $s0, 2
  get_element_address($a0, $a1)
  lb    $v0, ($v0)                  # get board[row-2][col]
  bne   $v0, hole, valid_return0
  j   valid_return1
valid_leftJump:
  blt   $s1, 2, valid_return0
  subi    $a1, $s1, 1
  get_element_address($a0, $a1)
  lb    $v0, ($v0)                  # get board[row][col-1]
  bne   $v0, peg, valid_return0
  subi    $a1, $s1, 2
  get_element_address($a0, $a1)
  lb    $v0, ($v0)                  # get board[row][col-2]
  bne   $v0, hole, valid_return0
  j   valid_return1
valid_bottomJump:
  bgt   $s0, 4, valid_return0
  # analogous to valid_topJump but replace subi with addi
valid_rightJump:
  bgt   $s1, 4, valid_return0
  # analogous to valid_leftJump but replace subi with addi
valid_return0:
  li    $v0, 0
```

```
  j   valid_return
valid_return1:
  li    $v0, 1
valid_return:
  # END GOES HERE
  jr    $ra
```

Code Block 2.9: Check if Valid Jump (MIPS)

2.3.5 Make Jump

`make_jump` uses the `modifyElement` macro to transform the source and intermediary peg into a hole, and the destination hole into a peg.

```
##### MAKE JUMP #####
  make_jump:
    # PREAMBLE GOES HERE
    move    $s0, $a0              # s0 = row
    move    $s1, $a1              # s1 = col
    move    $s2, $a2              # s2 = dir
    modifyElement(hole)           # board[row][col] = '.'
    beq   $s2, 1, make_leftJump
    beq   $s2, 2, make_bottomJump
    beq   $s2, 3, make_rightJump
  make_topJump:
    move    $a1, $s1
    subi    $a0, $s0, 1
    modifyElement(hole)           # board[row-1][col] = '.'
    subi    $a0, $s0, 2
    modifyElement(peg)            # board[row-2][col] = 'o'
    j   mjump_return
  make_leftJump:
    move    $a0, $s0
    subi    $a1, $s1, 1
    modifyElement(hole)           # board[row][col-1] = '.'
    subi    $a1, $s1, 2
    modifyElement(peg)            # board[row][col-2] = 'o'
    j   mjump_return
  make_bottomJump:
    # analogous to make_topJump but replace subi with addi
  make_rightJump:
    # analogous to make_leftJump but replace subi with addi
  mjump_return:
    move    $t6, $a0
    move    $t7, $a1
    lb    $t2, 2($gp)
    subi    $t2, $t2, 1           # decrement num of pegs
    sb    $t2, 2($gp)
    # END GOES HERE
    jr    $ra
```

Code Block 2.10: Make Jump (MIPS)

2.3.6 Solve Peg Solitaire

`peg_solve` is an extension of the solver function implemented in C code as it now contains the storing of paths in the path array. The pushing of paths is achieved by the `push_path` macro, which stores the path string `r,c->r',c'` character by character. On the other hand, the popping of paths implemented here is actually an *overwriting* of paths. That is, if we were to backtrack in the solver function, then we first bring the final path pointer 8 bytes backward. The consequence of this is that should no valid solution exist, then the path array will remain populated by unused paths which would not be printed to the console.

```
peg_solve:
  # PREAMBLE GOES HERE
  lb    $t0, 2($gp)                   # get total num of pegs
  li    $s0, 0                        # currRow = 0
  bne   $t0, 1, solve_loopRow
##### BASE CASE #####
solve_correctLoc:                     # if pegs = 1
  lb    $a0, 0($gp)                   # get final X coord
  lb    $a1, 1($gp)                   # get final Y coord
  get_element_address($a0, $a1)
  lb    $v0, ($v0)
  beq   $v0, peg, solve_return1       # if peg is at correct final loc
##### BASE CASE #####
solve_loopRow:
  beq   $s0, board_size, solve_return0  # if rows have been exhausted, return 0
  li    $s1, 0                        # currCol = 0
solve_loopCol:
  beq   $s1, board_size, solve_incrementRow
  li    $s2, 0                        # dir = 0
solve_loopDir:
  beq   $s2, 4, solve_incrementCol
  move    $a0, $s0
  move    $a1, $s1
  get_element_address($a0, $a1)
  lb    $v0, ($v0)
  bne   $v0, peg, solve_incrementCol  # return 0 if curr element is not a peg
  move    $a2, $s2
  jal   valid_jump                    # call valid_jump(row, col, dir)
  bne   $v0, 1, solve_incrementDir    # if jump is not valid, try next jump
  move    $a0, $s0
  move    $a1, $s1
  move    $a2, $s2
  jal   make_jump
  push_path()
  jal   peg_solve                     # recursive call
  beq   $v0, 1, solve_return1
##### BACKTRACK #####
solve_backtrack:
  subi    $t4, $t4, 8                 # move back path pointer by 8 bytes
  move    $a0, $s0
  move    $a1, $s1
  move    $a2, $s2
```

```
  jal   undo_jump
##### BACKTRACK #####
solve_incrementDir:
  addi    $s2, $s2, 1                 # move to next jump direction
  j   solve_loopDir
solve_incrementCol:
  addi    $s1, $s1, 1                 # move to next column
  j   solve_loopCol
solve_incrementRow:
  addi    $s0, $s0, 1                 # move to next row
  j   solve_loopRow
solve_return1:
  li    $v0, 1
  j   solve_return
solve_return0:
  li    $v0, 0
solve_return:
  # END GOES HERE
  jr    $ra
```

Code Block 2.11: Solve Peg Solitaire (MIPS)

2.3.7 Driver Program

The driver program uses an additional **init_board** function to initialize the peg solitaire board. It is also able to print the strings in the path array by using a start and end pointer for the array.

```
main:
  jal     init_board          # initialize peg board
  jal     find_dest           # find final destination of peg
  jal     count_pegs          # count current number of pegs
  la      $t4, path           # final pointer to path array
  jal     peg_solve           # call peg solver function
  beqz    $v0, main_fail
main_success:
  print(yes)                  # if return != 0, print 'YES'
  la      $t3, path           # start pointer to path array
main_printPath:
  beq     $t3, $t4, main_end  # if start ptr == final pointer
  newline()
  li      $t0, 0
main_pathInner:
  beq     $t0, 8, main_printPath
  lb      $a0, ($t3)
  li      $v0, 11
  syscall
  addi    $t3, $t3, 1
  addi    $t0, $t0, 1
  j       main_pathInner
main_fail:
  print(no)                   # if return == 0, print 'NO'
```

```
main_end:
  li      $v0,10              # finish execution
  syscall
```

Code Block 2.12: Count Pegs (MIPS)

Part 3

Testing

Three test cases are provided in this section along with their running time on a 64-bit Windows PC with an i5-8300H CPU @ 2.30GHz.

3.1 Trivial Board

Input

```
xx...xx
xx...xx
.......
..O....
.......
xx...xx
xx...xx
```

Output

```
YES
```

Execution Time: $\approx 0.9s$

3.2 Board with Maximum Pegs

Input

```
...o...
...o...
oo.....
.....oo
.......
..oo...
....E..
```

Output

```
YES
1,4->3,4
3,1->3,3
```

```
3,3->3,5
4,7->4,5
3,5->5,5
6,3->6,5
5,5->7,5
```

Execution Time: $\approx 0.9s$

3.3 Maximally Difficult Board

Input

```
.......
.......
.oo.oo.
.oo.oo.
.......
.......
....E..
```

Output

```
NO
```

Execution Time: $\approx 8.0s$