Lab # 8, Due: November 20, 2012

Overview

Peg solitaire is a board game played on a board that contains a collection of peg-holders (holes). Some of them have a peg on them. The goal is to remove all except one peg from the board. The rule for removing a peg is as follows: for any three holes that are consecutive adjacent holes A, B and C, if A and B have a peg and C is empty, then the pegs can be removed from positions A and B, and a peg placed at C. Each move, thus, removes exactly one peg.

One of the common board shapes is shown below:

This puzzle has been around for several centuries and has received the attention of some of the most famous mathematicians such as Euler. To get a first-hand experience of the game, you can play an electronic version of it. One web site that contains an applet for peg solitaire is http://www.cut-the-knot.org/proofs/pegsolitaire.shtml

Goals of the project:

- learn to implement a recursive, backtracking algorithm which is a powerful general purpose technique for a wide-range of searching problem.
- Learn to improve the performance of a backtrack algorithm using a hash table.
- Learn to use secondary data structures like arrays, lists and stacks to represent and maintain move sequences, board position, possible moves from a given board position etc.

You are to write a program that finds a solution (if it exists) with a given starting position. Your final output will compare two different implementations – one that implements the backtracking algorithm, and a second one that uses a hash table to speed-up backtrack searching. Some of you will implement closed-hashing and others will implement open-hashing.

Backtracking algorithm

General background on backtracking will be discussed in the lab. The backtracking algorithm for peg solitaire is shown below:

```
boolean move (board x, moveList mseq) {
if (solved(x)) return true;
curMoves = currentMoves(x); // set of all current
moves
for (every m in curMoves) {
   y = makeMove(x, m);
   if (move(y, mseq)) {
     mseq = push(m, mseq);
     return true;
   }
 }
return false;
}
```

Use of Hash table to avoid redundant searches

Backtracking algorithm as described above will solve starting positions with around 20 pegs. But for boards with more pegs, backtracking will take several minutes to several hours (even days!). Adding a Hash Table improves the performance of the algorithm significantly.
The key idea is as follows: Suppose X is the starting position. It is likely that two different move sequences s
and s2 will take X to the same board position Y. Backtracking algorithm will follow the path corresponding to sequence s1 and will call itself recursively with Y as input.

Suppose that Y does not lead to a solution. So this call will return false and so the search will continue. At a future point, the backtracking algorithm will follow the path s2 and will arrive at the same board Y and will make a call with Y as input for a second time. (This is the compound interest rule discussed in Chapter 2 of the text.) This redundancy can be removed by keeping track of all the boards for which the recursive call fails in a suitable data structure. The operations that should be (efficiently) supported by this data structure are search and insert. In this project, a ***hash table*** will be used for storing the boards.

Additional Data Structures

Choice of appropriate data structures for representing the board, moves, sequence of moves etc. will be discussed in the lab. It turns out that clever and complex data structures for these representations will not make a significant difference in the overall performance so we will use a simple data structure for these – to make the coding as simple as possible. Some details are provided below.

The size of the hash table should be carefully chosen (by experimentation). The following is a modified version of backtracking algorithm that uses a hash table.

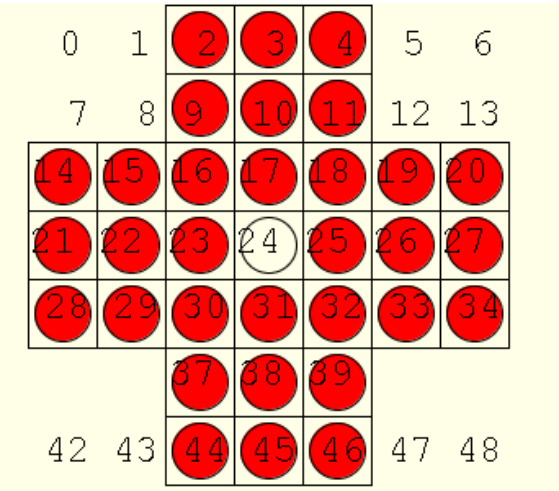
Modified backtracking

```
boolean move (board x, moveList mseq, Hashtable H) {
if (solved(x)) return;
curMoves = currentMoves(x); // set of all current moves
for (every m in curMoves) {
   y = makeMove(x, m);
   if (y is not in H) {// begin if 1
    if move(y, mseq) { // begin if 2
     mseq = push(mseq, m);
     return true;
     }
    else
    {insert y into H;}// end if 2
   } // end if 1
 }// end for
return false;
}
```

The only changes we have made are the following: (a) before making a recursive call, perform a search for the board and (b) insert the board into the hash table when the recursive call returns false.

Some implementation details

One way to represent the peg-solitaire board is as a 7 by 7 square in which the four 2 x 2 corners have been removed. See the figure below.



So the board can be represented as a boolean array of size 49 in which 1 represents a peg, 0 represents an empty slot. (slots like 0, 1, 5, 6, 7, 8 etc. will always be 0.

A move will be represented as a triple (vector of size 3) – such as (2, 3, 4) or (16, 9, 2).

In the above board, the total number of possible moves is 78 and the list of all such moves is given below:

(14 21 28)
(15 22 29)
(19 26 33)
(20 27 34)
(2 9 16)
(9 16 23)
(16 23 30)
(23 30 37)
(30 37 44)
(3 10 17)
(10 17 24)
(17 24 31)
(24 31 38)
(31 38 45)
(4 11 18)
(11 18 25)
(18 25 32)
(25 32 39)
(32 39 46)

(2 3 4)
(9 10 11)
(37 38 39)
(44 45 46)
(14 15 16)
(15 16 17)
(16 17 18)
(17 18 19)
(18 19 20)
(21 22 23)
(22 23 24)
(23 24 25)
(24 25 26)
(25 26 27)
(28 29 30)
(29 30 31)
(30 31 32)
(31 32 33)
(32 33 34)
(28 21 14)
(29 22 15)
(33 26 19)
(34 27 20)
(16 9 2)
(23 16 9)
(30 23 16)
(37 30 23)
(44 37 30)
(17 10 3)
(24 17 10)
(31 24 17)
(38 31 24)
(45 38 31)
(18 11 4)
(25 18 11)
(32 25 18)
(39 32 25)
(46 39 32)
(4 3 2)
(11 10 9)
(39 38 37)
(46 45 44)
(16 15 14)
(17 16 15)

(18 17 16)
(19 18 17)
(20 19 18)
(23 22 21)
(24 23 22)
(25 24 23)
(26 25 24)
(27 26 25)
(30 29 28)
(31 30 29)
(32 31 30)
(33 32 31)
(34 33 32)

You can hard-code this table of all possible moves in your implementation.

Some Test Cases and Results:

Shown below are three test cases - two of which have solutions and one does not. When a solution exists, the output shown is the sequence of moves (in reverse order) to reach the winning position.

Board 1: (9 18 19 29 30 38) – the numbers indicate the positions of the pegs.

Solution: (9 10 11) (24 17 10) (19 18 17) (38 31 24) (29 30 31)

Board 2: (2 3 4 9 10 14 15 16 17 19 20 21)

Solution: No solution

Board 3: (2 3 4 9 10 14 15 16 17 19)

Solution: (2 3 4) (17 10 3) (19 18 17) (4 3 2) (16 17 18) (2 9 16) (23 16 9) (14 15 16) (9 16 23)

The test cases with around 12 pegs would run in a few minutes without hashing. Note that, in general, the board positions with no solution will take much longer than a board position (with the same number of pegs) that has a solution. When the board has more than 15 pegs, the plain backtracking version will take a long time (possibly several hours). For such boards, hash table based backtracking will provide a solution much faster. The following 20 peg board was solved using an open-hashing based backtracking program in about *10 seconds*:

Input: (3 4 9 15 16 17 19 20 22 23 25 26 27 30 31 33 34 38 45 46)

Solution: (32 39 46) (34 33 32) (25 32 39) (27 26 25) (30 31 32) (16 23 30) (18 17 16) (4 11 18) (25 18 11) (20 19 18) (29 30 31) (44 37 30) (46 45 44) (3 10 17) (24 17 10) (38 31 24) (9 16 23) (23 30 37) (15 22 29)

(Note that the sequences output by your program need not be exactly as above. It may find a different solution depending on how the tree is searched.

How to generate test cases?

To generate a board (with a specified number of pegs) that is not solvable, you can randomly place pegs and try. (There is a good chance that a randomly generated board has no solution.)

To generate a board with a specified number of pegs that has a solution, you can start with a board with one peg and start making moves in reverse until you reach the board with the desired number of pegs. You can use the Java applet in the web page http://www.mazeworks.com/peggy/index.htm to make the moves in reverse. Set the option to reverse and choose "solitaire" as the end goal. Here is a board position I created by this process.

Expected program functionality

When the program is run, it should ask for a board input. The board will be represented as a sequence of peg positions stored in a file. Then, the program runs the backtrack search algorithm using (a) no hash table (b) open hashing and (c) closed hashing (double

hashing) and report output in all the three cases. It should also output the CPU time in each case. The outputs should be displayed on the console as well as written to files specified as argv file names.

Typical inputs for which your program will be tested will have 15 to 25 pegs.

What should be submitted?

Please include your source code, and all the supporting source files (for example, the code that implements hash tables), and compiled code along with at least two sample input files and the corresponding outputs written to files.