CS 315 Data Structures

Fall 2008

Project # 4, Due: November 14, 2008

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 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.
- to enhance the performance of a backtrack algorithm using additional data

structures, in this case, a hash table.
- Learn to use some 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 three different implementations – one that directly implments the backtracking algorithm, and the other two based on a speed-up by memorizing previously solved board positions (known technically as memoization) using a hash table. The two different solutions are based on open hashing and closed 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;

curMoves = currentMoves(x); // set of all current moves

for (every m in curMoves) {

   add m to mseq;

   y = makeMove(x, m);

   if move(y, mseq) return true;

   else

    remove m from mseq;
 }
return false;
}
```

Use of Hash table to avoid redundant searches

The key idea behind this improvement is as follows: Let X be the starting board position. It is conceivable (even quite likely) that two different move sequences s1 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 the 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 so the call with Y as input will be run another time. This redundancy can be removed by keeping track of all the boards that have been attempted (and failed) in a suitable data structure. The operations that should be 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

The question of choosing the appropriate data structures for representing the board, a move, and a sequence of moves etc. will be discussed in the lab. But it turns out that clever and complex data structures will not make significant difference in the overall performance so a simple choice that simplifies coding is the best way to address this choice.

The size of the hash table should be carefully chosen. Note that the savings in redundant search comes at a price – the overhead associated with the maintenance of the hash table. So a careful choice of the hash table size is important.

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) {

  if m is not in H {

    add m to mseq;

    y = makeMove(x, m);

    if move(y, mseq) return true;

    else

     {insert y into H; remove m from mseq;}
  }

 }

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.

Variations to be experimented with

1) selective insertion of board – instead of inserting all the failed boards, you can selectively insert – for example, all those with a certain fixed number of pegs in them.

2) try different hash functions – based on multiplication method, division method etc. Also the size of the hash table size should be determined experimentally.

3) the order in which the for loop is explored plays a crucial role in the performance of the algorithm. For example, in the case of queens problem, using row or column order does not perform very well – a random choice of order performs much better. The same may be true for this problem as well.

What should be submitted?

When the program is run, it should ask for a board input. The board will be represented as a 33 bit string. 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.

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