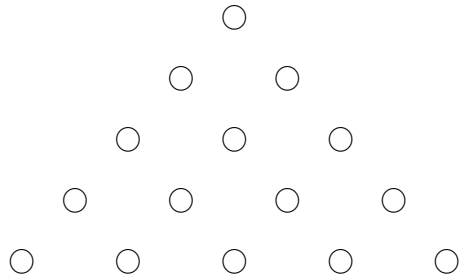
Solving an IQ ("Cracker Barrel") puzzle

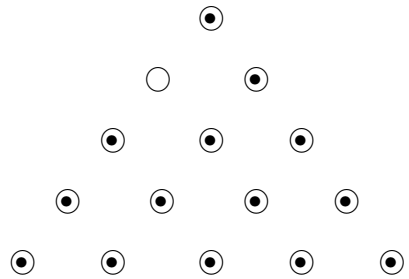
Progamming Languages 4003§2 programming assignment #5

`clingo` program due 3:30 PM, Thor's Day 31 March 2016

The puzzle is played on a triangular board with 15 holes:

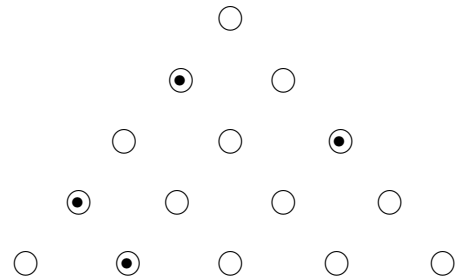


And you are give 14 pegs to put into 14 of the 15 holes. You, the solver, can pick which hole to leave blank. Say you have



A legal move consists of picking one peg and jumping it over *one* adjacent peg into an empty hole. Then you take the peg you peg you jumped over off the board. So, for example, in the board above there are 2 legal first jumps: (i) the peg in row 4 column 1 can jump over the peg in row 3 column 1 to the hole in row 2 column 1; and (ii) the peg in row 4 column 3 can jump over the peg in row 3 column 2 into the hole in row 2 column 1. Note that there are 6 directions to move — in each of 2 ways parallel to one of the 3 edges.

The goal is to make 13 legal moves and end up with only 1 peg left on the board. If you play at random, usually you end up with > 1 peg on the board but no more possible legal moves, as in



Write a `clingo` **program** to solve the problem.

Rules & Regulations: (You knew I'd give you some.)

1. Number the positions by row and column, as I did above. Basically, think of the board as

	column				
row	1	2	3	4	5
1	◯				
2	◯	◯			
3	◯	◯	◯		
4	◯	◯	◯	◯	
5	◯	◯	◯	◯	◯

   And there the 6 directions to hop are north (↑), south (↓), east (→), west (←), northwest (↖), and southeast (↘).

2. Your program should define the rules and of the game and the goal of the game and then depend upon `clingo` to find a solution; an answer set should correspond to a single solution.
3. Count times as the number of moves made, so they'll range from 0 to 13.
4. Your program *must* define 2 relations,
   (a) "`occupied(Time,Row,Clmn)`" which tells which holes have pegs in them at which times.
   (b) "`nextJump(Time, FromRow,ClmFrom,ToRow,ToClmn)`", which says, at time `Time`, that your *next* move is to jump the peg in hole `(FromRow,FromClmn)` to hole `(ToRow,ToClmn)`.
5. And your program contain the 2 lines "`#show occupied/3.`" and "`#show nextJump/5.`".

   You must use *exactly* these relation names, number of parameters, and meanings. Of course, what you call the parameters is your decision, as long as your program is clear to read (in my opeinion). Also, of course, you may define as many additional relations as you choose — again, subject to having a readable program. (For example, in my program logic, I didn't happen to use the relation `nextJump` above. Instead I defined separate `jumpFrom` and `jumpTo` relations. So to meet this requirement I'd add the extra rule

```
nextJump(Time,FromR,FromC,ToR,ToC)
    :- jumpFrom(Time,FromR,FromC), jumpTo(Time,ToR,ToC).
```

   to define relatoin `nextJump` from `jumpFrom` and `jumpTo`. You're free to do the same.)

In case you're interested: Run your program to get, say, 10 or 20 answer sets. Do they all agree on the first jumps, or on the lasts? That will tell you something about `clingo`'s (actually, `clasp`'s) heuristics for what to try next as it searches for a solution.

For a <u>small amount</u> of extra credit: Write a JAVA program to take your output (for 1 answer set) and do a visual representation of the game. The winner will be expected to explain his program in class. (To be eligible for any extra credit, you must turn in your JAVA program on time — no late extensions on this part.)

A challenge for later (to which I do not know an answer and which, I expect, cannot be handled in `clingo`)**:** Find a ***<u>simple</u>*** set of rules that tells you, if you are given a board with some of the moves already made, what move to make next to (more or less) optimize your chances of winning.