**3.1.**

For the traditional board of English and European, both of them are square board without four corners. I could use X dimension and Y dimension to represent. It is also enough to represent the four moving directions, up, down, left, right. However, in order to make it easier, I used location predicate with two arguments to replace X dimension and Y dimension. The predicate is location. It could be written as location(X, Y). X is the number of the X dimension and Y is the number of the Y dimension.

For the hexagonal board as follows (Figure 1) and triangular board (Figure 2), the pegs have six possible directions to move rather than four directions in the traditional board. The six directions are left, right, top left, top right, bottom left, bottom right. Therefore, the layout representation is more complicated than the traditional board. I thought that odd and even number could be used to represent the layout. For example, looking at the board in figure 3, there are seven lines from 1 to 7. For odd lines, holes could be represented by odd number sequence. So, for even lines, holes could be represented by even number sequence. We could see the regularity clearly. I drew a picture to show it in Figure 3. It also could be written as location(X, Y).

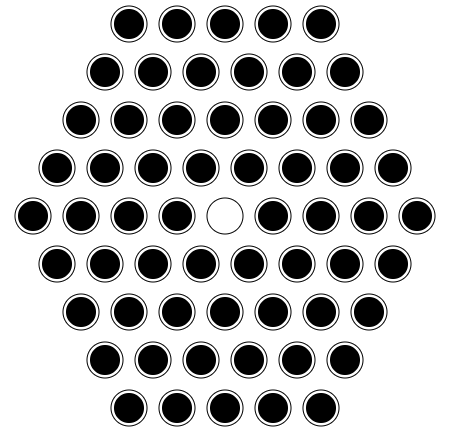


Figure 1

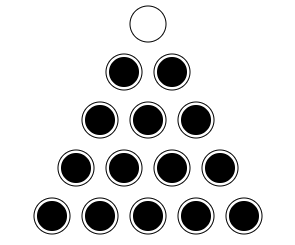


Figure 2

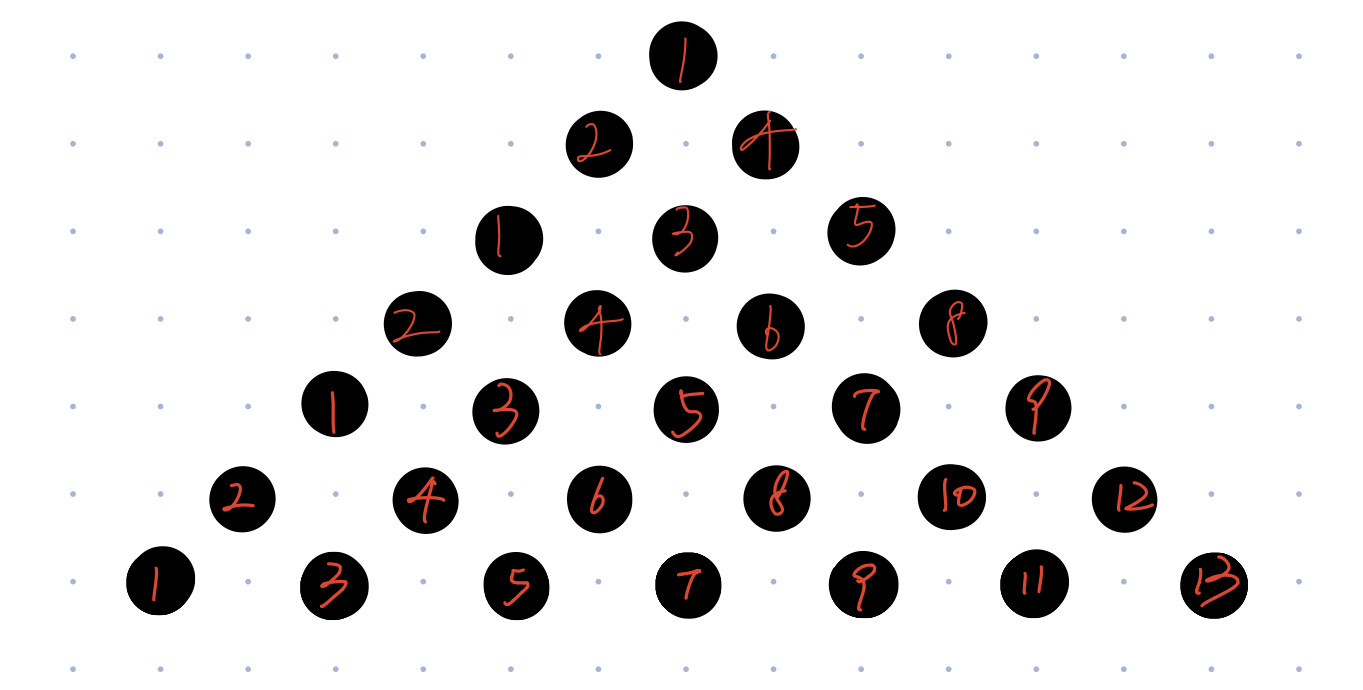


Figure 3.

**3.2.**

The layout of classical 33-holes English-style puzzle is as follows.

location(1..2, 3..5).

location(3..5, 1..7).

location(6..7, 3..5).

**3.3.**

**An instance**

In the initial state of the board, there is only one empty hole and other holes are occupied, Therefore, I used occupied predicate and empty predicate. The final state use :- state to represent the final empty and occupied state.

**A solution**

For each occupied hole, the peg could move to different directions. Direction represents the possible moving directions, up, down, left, right. I also used a couldmove predicate to show which location could move to which direction. Furthermore, because I need to move the peg step to step, I thought I need a timestep predicate. I set the t time step and I want to get the final states at the t+1 time step. Therefore, the movement at each time step could be represented by move predicate, the exclude the impossible couldmove location. For each location at each time step, it has a unique state. So, after moving from one timestep to the next timestep, the states of corresponding locations should be changed and other states of locations should not be changed. In the end, I intended to show the finish states. Then, I could use show to output the state and move finally. The solution predicates are as follows: timestep, state, couldmove, move, changed.

**3.4.**

I think that you need to add the layout basing on what I said before and change the code of the initial states and final states as follows. I add some comments which could help you find the code of defining the initial states and final states and understand my code. Noting that you also need to define the time step which you expect to get the final state at t+1 time step.

%%% location is occupied.

occupied(X, Y) :- location(X, Y).

%%% location is empty.

empty(X, Y) :- location(X, Y).

%%% Timestep. You need to set the expected time step which you want to finish the process and get the final state at t+1 timestep.

timestep(1..t).

%%% Declare the initial state of each location. state(occupied/empty(X, Y), timestep(T)). 1 is the initial time step.

%state(occupied(1, 1), 1).

%state(occupied(2, 1..3), 1).

%state(empty(3, 1), 1).

state(occupied(4, 3), 1).

state(empty(1, 2), 1).

state(occupied(2, 2..4), 1).

state(occupied(3, 1..3), 1).

%%% Direction.

direction(up). direction(down). direction(left). direction(right).

%%% move one peg at each timestep.

1 { move(D, X, Y, T) : direction(D), occupied(X, Y) } 1 :- timestep(T).

%%% Movement. What peg could move to which direction and location at T time step.

couldmove(up, X, Y, T) :- state(occupied(X, Y), T), state(occupied(X, Y-1), T), state(empty(X, Y-2), T), timestep(T).

couldmove(down, X, Y, T) :- state(occupied(X, Y), T), state(occupied(X, Y+1), T), state(empty(X, Y+2), T), timestep(T).

couldmove(left, X, Y, T) :- state(occupied(X, Y), T), state(occupied(X-1, Y), T), state(empty(X-2, Y), T), timestep(T).

couldmove(right, X, Y, T) :- state(occupied(X, Y), T), state(occupied(X+1, Y), T), state(empty(X+2, Y), T), timestep(T).

% exclude could not move.

:- move(D, X, Y, T), not couldmove(D, X, Y, T), timestep(T).

%%% change the state from current time step to the next time step if the location could move.

state(occupied(X, Y-2), T+1) :- move(up, X, Y, T), timestep(T).

state(empty(X, Y-1), T+1) :- move(up, X, Y, T), timestep(T).

state(empty(X, Y), T+1) :- move(up, X, Y, T), timestep(T).

state(occupied(X, Y+2), T+1) :- move(down, X, Y, T), timestep(T).

state(empty(X, Y+1), T+1) :- move(down, X, Y, T), timestep(T).

state(empty(X, Y), T+1) :- move(down, X, Y, T), timestep(T).

state(occupied(X-2, Y), T+1) :- move(left, X, Y, T), timestep(T).

state(empty(X-1, Y), T+1) :- move(left, X, Y, T), timestep(T).

state(empty(X, Y), T+1) :- move(left, X, Y, T), timestep(T).

state(occupied(X+2, Y), T+1) :- move(right, X, Y, T), timestep(T).

state(empty(X+1, Y), T+1) :- move(right, X, Y, T), timestep(T).

state(empty(X, Y), T+1) :- move(right, X, Y, T), timestep(T).

%%% declare unchanged states by define unchanged locations

%%% show changed location at T+1 time step since unchanged location are difficult to show directly.

changed(X, Y-2, T+1) :- move(up, X, Y, T), timestep(T).

changed(X, Y-1, T+1) :- move(up, X, Y, T), timestep(T).

changed(X, Y, T+1) :- move(up, X, Y, T), timestep(T).

changed(X, Y+2, T+1) :- move(down, X, Y, T), timestep(T).

changed(X, Y+1, T+1) :- move(down, X, Y, T), timestep(T).

changed(X, Y, T+1) :- move(down, X, Y, T), timestep(T).

changed(X-2, Y, T+1) :- move(left, X, Y, T), timestep(T).

changed(X-1, Y, T+1) :- move(left, X, Y, T), timestep(T).

changed(X, Y, T+1) :- move(left, X, Y, T), timestep(T).

changed(X+2, Y, T+1) :- move(right, X, Y, T), timestep(T).

changed(X+1, Y, T+1) :- move(right, X, Y, T), timestep(T).

changed(X, Y, T+1) :- move(right, X, Y, T), timestep(T).

%%% define unchanged states by unchanged location.

state(occupied(X, Y), T+1) :- state(occupied(X, Y), T), not changed(X, Y, T+1), location(X, Y), timestep(T).

state(empty(X, Y), T+1) :- state(empty(X, Y), T), not changed(X, Y, T+1), location(X, Y), timestep(T).

%%% finsh state, t time step is the last moving time step so the t+1 time step is the final time step.

%:- not state(empty(2, 1..3), t+1).

%:- not state(empty(3, 1), t+1).

%:- not state(occupied(1, 1), t+1).

:- not state(occupied(4, 3), t+1).

:- not state(empty(1, 2), t+1).

:- not state(empty(2, 2..4), t+1).

:- not state(empty(3, 1..3), t+1).

%%% Show.

#show move/4.

#show state/2.