Brandon Druschel CSC 466
2/21/19

## Project Title:

**Peg Solitaire Heuristic Learning Machine**

## Project Description:

A rule-based Peg Solitaire self-playing machine will be developed using genetic algorithms. The machine will play a large amount of games using both heuristically as well as randomly determined moves. The machine will continue this process until it manages to win the game (get one peg remaining on the board). The machine will be assessed by recording its performance during each game (i.e. how many pegs were left in the end), which will represent the fitness metric.

## Task Decomposition

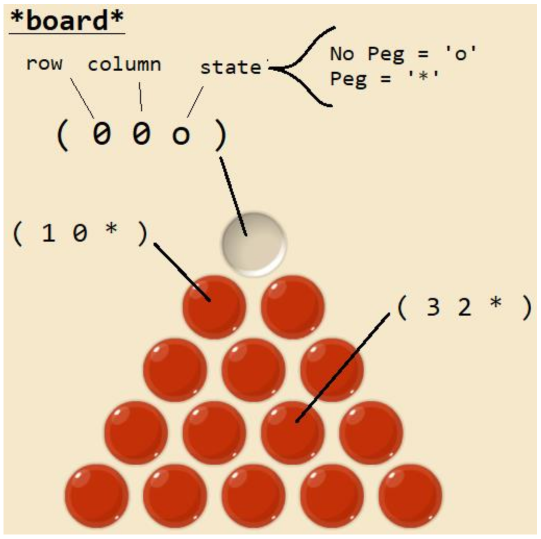
1. **Global List**

   Description:

   Start by creating a file called **pegs.l**, which will serve as the main program. Have a global list called ***moves***, which will serve as the list of all possible moves a player may make on a triangular board, i.e., any sort of "jumps" that can be made with a single peg from its original position, over a neighboring peg, into an empty space.

   Create another global list called ***board*** which lists every possible coordinate within the triangular board and whether that has a peg or not, i.e., `'(0 0 o)'` is the top corner of the triangle – which is an empty hole, while `'(1 0 *)'` and `'(1 1 *)'` represent the first row down – each with pegs, etc.

   Have a **reset** method which simply sets the *board* list back to its beginning state.



   Demo:

```
[]> *moves*
( L R UL UR DL DR ) ;; left, right, up-left, up-right, down-left, down-right
[]> *board*
… a list representing a whole triangular board in its beginning state.
[]> ( reset )
… the board is set back to its beginning state.
```

2. **Random Move Generator**
   Description:
   Create a **`pick`** method which selects one of the **`*moves*`** at random and returns it. Then, create a **`play`** method will take an integer x as input and use the `pick` method to generate a randomized list of x number of moves.

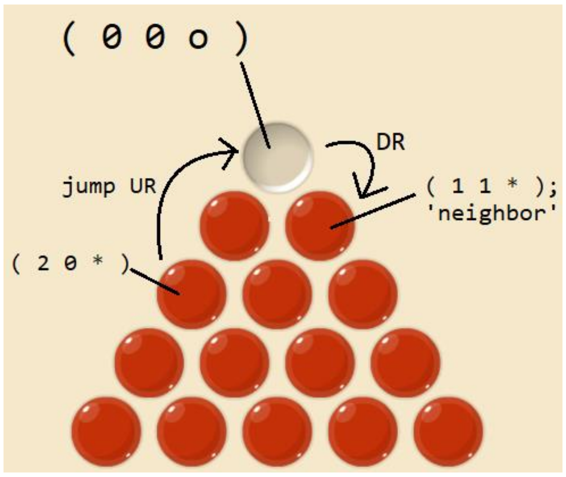
   Demo:
```
[]> ( pick )
DL
[]> ( pick )
R
[]> ( pick )
UR
[]> ( play 5 )
( DL R UR )
[]> ( play 12 )
( L UL L DR DL R UL DL DR L R UR )
```

3. **Draw a Peg Board**
   Description:
   Have a **`visualize`** method which analyzes the `*board*` list and provides a 2D visualization of a triangular board in its current state. Let **'o'** represent an open space and let **`'*'`** represent a peg.

   Demo:
```
[]> ( visualize )
-- GAME BOARD –-
    o
   * *
  * * *
 * * * *
* * * * *
```
**`(Assuming the board is at its beginning state)`**

4. **Labels & Constraints**
   Description:
   The rules/constraints of Peg Solitaire are to be determined using several “boolean”-like methods. The methods should be written in a manner that respects the constraints given by a *triangular board* game of Peg Solitaire. The other methods will help with identifying important elements of the game, such as the *neighbor* of a particular position, the position in which a jumped peg would end up, etc.
   - **`pegp`** takes an element from `*moves*` and returns **`t`** if it has a peg, else **`nil`** if it’s an empty hole

- **neighbor** takes a *board* position and a *moves* element, and returns the position where the *jumped peg* would be sitting. I.e., the neighboring peg.
- **jump** takes a *board* position and a *moves* element, and returns the position where the peg would end up, depending on the given move.
- **jumpp** takes a *board* positon and an element from *moves*, returning t if a free space is available in the position the peg jumps to.
- **count** looks at the *board* and returns how many pegs are left



Demo:

```
[]> ( pegp ‘(0 0 o) )
NIL
[]> ( pegp ‘(1 0 *) )
T
[]> ( neighbor ‘(0 0 o) DR )
(1 1 *)
[]> ( jump ‘(2 0 *) UR )
(0 0 o)
[]> ( jumpp ‘(0 0 o) DR )
NIL
[]> ( jumpp ‘(2 0 *) UR )
T
[]> ( count )
14 ← Assuming the board is at its beginning state.
```

5. **Determining a Valid Move**

Description:

The **validate** method takes a spot from the *board* and an element from *moves* and determines if the such a move can be done.

I.e. on a starting board, **( validate ‘(2 0 *) ur )** is an acceptable move (will return 't') since:

- (2 0 *) actually exists within the *board*
- (2 0 *) contains a peg
- The move UR leads you to an empty hole; (0 0 o)
- There is a peg present between the two spots; (1 0 *)

If any one of these conditions are violated in some way, then the move is deemed 'invalid' (return NIL).

Demo:

```
[]> ( validate ‘(0 0 o) DR )
NIL
[]> ( validate ‘(2 0 *) UR )
T
```

6. **Make Your Move**
   Description:
   It seems we're finally ready to create an actual **move** method. This method will take a position from the `*board*` and a `*moves*` element and, ***assuming the move is legal***, apply the move to the `*board*`. Thus, the state of the `*board*` will be modified according to the move that is made; the peg will move from its starting position to the end position, and the neighboring peg will be removed from the game board.
   Now, you'll want to update your **play** method to pick a random move *and* a random spot on the `*board*` to apply it to – if the move is illegal, run the method again. You're also encouraged to write a separate play method which takes no parameters, and forces more moves into the play until no remaining moves exist.

   Demo:
```
[]> ( visualize )
-- GAME BOARD –-
    o
   * *
  * * *
 * * * *
* * * * *
[]> ( move ‘(2 0 *) UR )
NIL
[]> ( visualize )
-- GAME BOARD –-
    *
   o *
  o * *
 * * * *
* * * * *
[]> ( play 3 )
( ((2 0 *) UR) ((3 2 *) UL) ((3 0 *) R) )
```

7. **State of the Game**
   Description:
   The last step in having a playable game of Peg Solitaire involves determining the end states of the game.
   Write a method called **goalp** which determines whether the goal state (only one peg remaining) has been reached.
   The last method, **failp**, which scans the remaining pegs on the `*board*` and determines whether there are any possible moves left. (Hint: Use the `validate` method)

   Demo:

```
[]> ( visualize )
-- GAME BOARD --
    o
   * *
  * * *
 * * * *
* * * * *
[]> ( goalp )
NIL
...
[]> ( visualize )
-- GAME BOARD --
    o
   o o
  o o o
 * o o o
o o o o o
[]> ( goalp )
T
...
[]> ( visualize )
-- GAME BOARD --
    *
   o o
  o o *
 o o o o
o o o * o
[]> ( failp )
T
...
[]> ( visualize )
-- GAME BOARD --
    *
   * o
  o o *
 o o o o
o o * * o
[] ( failp )
NIL
```

8. **Mutation**

   Description:

   To begin the evolutionary process we will need a method to mutate the *play* list. The **`mutation`** method will accomplish this by replacing a random element of `play` with a different move, *then* changing any remaining moves afterwards that may no longer be legal.

   Demo:

```
[]> ( setf p (play 5) )
( ((2 0 *) UR) ((3 2 *) UL) ((3 0 *) R) ((0 0 *) DL) ((3 3 *) L) )
[]> ( mutate p )
( ((2 0 *) UR) ((3 2 *) UL) ((3 0 *) R) ((4 3 *) UL) ((3 3 *) L) )
```

9. **Crossover**
   Description:
   We will need a method to crossover two sets of *plays*. The **`crossover`** method will construct a string consisting of the first ‘n’ elements from one play followed by a randomly selected move from the other play, with the last (*length* - n) moves “reshuffled” to abide by the rules of the games.

   Demo:
```
[]> ( setf m (play 5) )
( ((2 0 *) UR) ((3 2 *) UL) ((3 0 *) R) ((0 0 *) DL) ((3 3 *) L) )
[]> ( setf f (play 5) )
( ((2 2 *) UL) ((2 0 *) R) ((4 1 *) UR) ((0 0 *) DL) ((3 3 *) UL) )
[]> ( crossover m f )
( ((2 0 *) UR) ((3 2 *) UL) ((4 1 *) UR) ((0 0 *) DL) ((1 1 *) DL) )
;; In this demo, the 3rd move of ‘f’ is mutated into the 3rd move of ‘m’, with
;; the last move in ‘m’ reshuffled to a legal move
```

10. **Fitness Metric**
    Description:
    A simple fitness metric will be used to represent how well the machine performs in a game by counting the amount of remaining pegs on the board.

    Demo:
```
[]> ( setf x (play) )
( ((2 0 *) UR) ((3 2 *) UL) ((4 1 *) UR) ((0 0 *) DL) ((1 1 *) DL) ((3 0 *) UR
((3 3 *) UL) ((4 2 *) UL) ((4 4 *) L) ((2 0 *) UR) ((0 0 *) DR) )
[]> ( fitness x )
3
```

11. **Individual Class**
    Description:
    Within the framework of GA computation lies the ‘individuals’. Here you will model the individual class and its accompanying methods using Task 6 of the RBG GA as a reference. In this case, the individual class will have *play*, *fitness*, and *number* as its variables.

12. **Population Class**
    Description:

Another element of the GA framework is the 'population' class. Here you will model the population class using Task 7 of the RBG GA as reference.

13. **Incorporate Mutation**
    Description:
    Mutation operators applicable to individual objects will come into play when implementing the *copy* and *crossover* operators. This step will work on the incorporation of mutation into the code. Use Task 8 of the RBG GA as reference.

14. **Copy**
    Description:
    Perhaps the most basic genetic operator in the context of evolutionary programming, "Copy" amounts to a favored random selection of an individual from the source population, followed by a drop of the selected individual, sometimes after mutation, into the destination population. Use Task 9 of the RBG GA as reference.

15. **Crossover (2)**
    Description:
    The genetic operator of "crossover" selects two favored individuals from the current population, creates a new individual by taking elements of each favored individual (i.e., by performing crossover), possibly mutates the newly created individual, then adds it to the new population. Use Task 10 of the RBG GA as reference.

16. **The GA**
    Description:
    The GA itself puts it all together! Use Task 11 of the RBG GA as reference… and hope to god it works!!

**(NOTE: The project would normally have had 15 tasks, but Task 1 was split into two equally simple tasks by Graci's request)**