Peg Solitaire Learning Machine

By Brandon Druschel

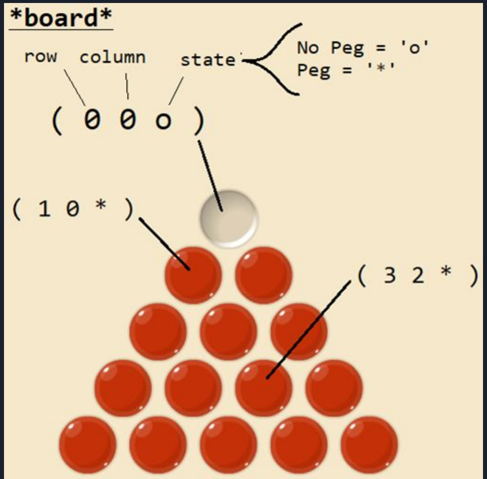
What is ‘Peg Solitaire’?

Example of a peg solitaire game being played on a ‘plus’-(+) shaped board →

- **Peg solitaire** is a board game in which one player moves a set of pegs on a board with holes
- A standard game fills the entire board with pegs except for the central hole. The objective is to empty the entire board except for a single peg in the central hole
- To remove a peg, the player must move a peg orthogonally over an adjacent peg into a hole two positions away, then the jumped peg is removed
- If a legal move is no longer possible, the game ends
- For this project, I chose to represent the **triangular board** due to its simplicity and popularity in restaurant chains, e.g. Cracker Barrel

Implementing a Peg Board (Task 1-3)

- A single **position** on the board is represented as a *list* containing **(r c s)**
  - r = row coordinate; top to bottom
  - c = column coordinate; left to right
  - s = state; ‘ * ’ = peg and ‘ o ’ = no peg
- The **triangular board** itself is made of up a list of coordinates; a *list of lists*
- Movement directions : left (L), right (R), up-left (UL), up-right (UR), down-left (DL), down-right (DR)




```
( setf *board* '(       (0 0 o)
                     (1 0 *)  (1 1 *)
                (2 0 *)  (2 1 *)  (2 2 *)
           (3 0 *)  (3 1 *)  (3 2 *)  (3 3 *)
      (4 0 *)  (4 1 *)  (4 2 *)  (4 3 *)  (4 4 *)
    )
)
```

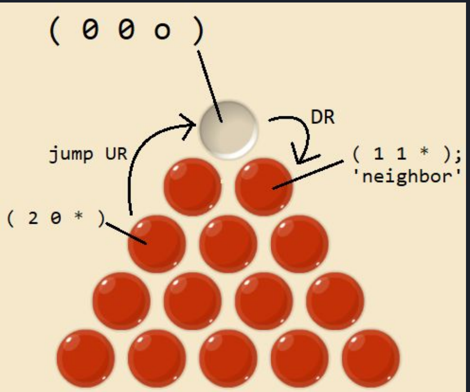
← The *board* list, formatted in such a way that is easily comprehensible

```
[2]> (visualize)
-- GAME BOARD--
    O
   * *
  * * *
 * * * *
* * * * *
```

← The current visualization of the board in the Lisp program

Creating a Playable Game (Task 4-7)

- The following **attributes** help define the rules of the game:
  - The **neighbor peg** is the peg to be jumped over (e.g. ‘(1 0 *)’ is the neighbor up-right of ‘(2 0 *)’)
  - The **jump position** is the position in which the peg would end up after it jumps over the neighbor peg. This position must be *empty* (s = ‘ o ‘)
  - A **peg count** simply tracks how many pegs remain on the board
- There are **two methods** which track whether the game is finished with *one* peg left (‘goalp’) or finished with *more than one* peg left (‘failp’). In both cases, each position is “scanned” to see if any more moves can be done in any position




```
[2]> (play-full)
-- GAME BOARD--
    o
   o o
  o o o
 o o o o
* o o o *
(((2 0 *) UR) ((4 0 *) UR) ((3 2 *) UL) ((4 1 *
 ((2 2 *) DL) ((4 2 *) L))
```

```
[3]> (play-full)
-- GAME BOARD--
    *
   o o
  * o o
 o o o *
o o o * o
(((2 0 *) UR) ((4 0 *) UR) ((3 2 *) L) ((3 0 *)
[4]>
```

↑ Two examples of the program playing a full game, with 2 and 4 pegs remaining respectfully. Note how no more legal moves can be done in either case.

Implementing Genetic Algorithm Attributes (Task 8-15)

- **Mutation** method
  - Change one random index to a different, legal move, and modify the rest of the moves accordingly.
- **Crossover** method
  - Change one random index in the *mother* individual to a move that is present in the *father* individual, and modify the rest of the moves accordingly
- **Fitness** metric
  - Simply the number of remaining pegs left on the board (for now)
  - Plan on implementing a second fitness metric based on distance between remaining pegs at the end of a game
- Rest of development was based closely around the **'RBG' genetic algorithm** from CSC 416
  - In particular, Tasks 6-11 of the RBG GA
  - Individual + Population Class
  - Incorporating mutation
  - Copy
  - Implementing crossover
- Final step (Task 16) in progress, which will bring it all together

Implementing Genetic Algorithm Attributes (Task 8-15)

- **Mutation** method

```
[2]> ( setf p (play-full) )
(((2 0 *) UR) ((3 2 *) UL) ((4 1 *) UR) ((1 0 *) DR) ((4 3 *) L) ((3 3 *) L)
((3 0 *) R) ((1 1 *) DR) ((4 0 *) R) ((3 3 *) L) ((4 2 *) UL))
[3]> ( setf p (mutate p) )
 (((2 0 *) UR) ((3 2 *) UL) ((4 1 *) UR) ((1 0 *) DR) ((4 3 *) L) ((4 0 *) UR)
((3 3 *) L) ((2 0 *) DR) ((1 1 *) DR) ((4 2 *) L) ((4 4 *) UL))
```

- **Crossover** method

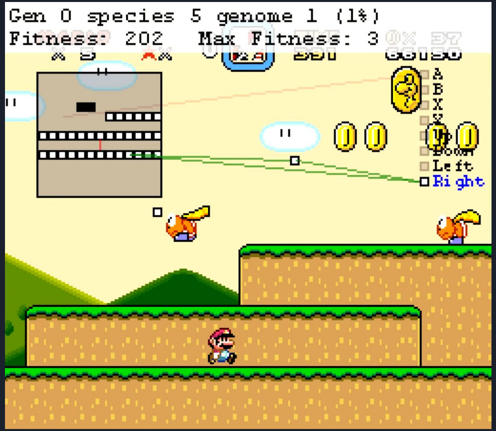
```
[2]> m
(((2 0 *) UR) ((3 2 *) UL) ((0 0 *) DL) ((3 0 *) R) ((3 3 *) L) ((1 1 *) DR)
((4 4 *) UL) ((4 2 *) R) ((2 0 *) DR) ((4 1 *) R) ((4 4 *) L))
[3]> f
(((2 0 *) UR) ((4 2 *) UL) ((1 1 *) DL) ((4 0 *) R) ((2 0 *) DL) ((4 2 *) UL)
((4 3 *) UL) ((3 3 *) UL)
 ((0 0 *) DR))
[4]> ( crossover m f )
(((2 0 *) UR) ((3 2 *) UL) ((0 0 *) DL) ((3 0 *) R) ((3 3 *) L) ((1 1 *) DR)
((4 4 *) UL) ((4 2 *) R) ((4 0 *) R))
[5]> ( crossover m f )
(((2 0 *) UR) ((3 2 *) UL) ((0 0 *) DL) ((3 0 *) R) ((4 3 *) UL) ((4 1 *) R)
((4 4 *) L) ((1 1 *) DL) ((2 2 *) DR))
```

- **Fitness** metric

```
[2]> (setf x (play-full) )
-- GAME BOARD--
    O
   O O
  O O *
 O O O O
* O * O O
(((2 0 *) UR) ((3 2 *) UL) ((0 0 *
 ((4 2 *) R) ((2 0 *) DR) ((4 1 *)
[3]> ( fitness x )
3
[4]> (setf x (play-full) )
-- GAME BOARD--
    O
   O O
  * * *
 O O O O
* O O O *
(((2 0 *) UR) ((4 2 *) UL) ((1 1 *
 ((3 3 *) UL) ((0 0 *) DR))
[5]> ( fitness x )
5
```

Inspiration - Why Use Genetic Algorithms to Optimize Game Playing?

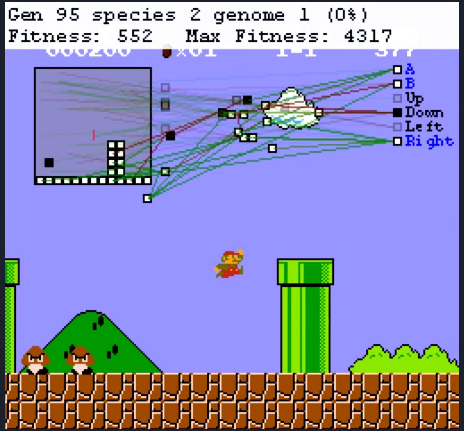
- Genetic algorithms have actually been used to play games since its inception
  - In 1963, **Nils Aall Barricelli** – considered a pioneer in artificial life research – simulated the evolution of the ability to play a simple game
- **'MarI/O'** is a machine learning program by SethBling that can complete a level in *Super Mario* games using neural networks and **genetic algorithms**
  - Specifically utilizes the "NEAT" algorithm; NeuroEvolution of Augmenting Topologies
  - Proof that genetic algorithms can be utilized successfully even in a video game



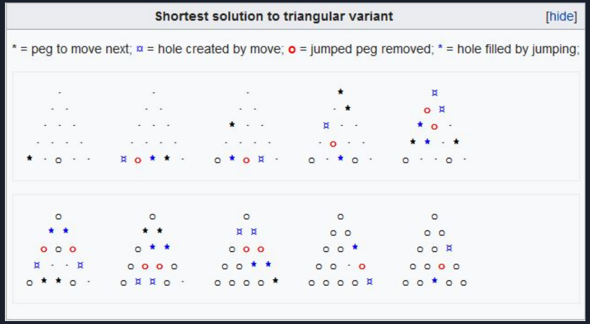
←

Examples of the 'MarI/O' program playing two separate Mario games. Note how the program tracks generations, fitness, and max fitness

→



Interesting Findings



- **The Pagoda function** is useful for showing the infeasibility of a given, generalized, peg solitaire problem.
  - A solution for finding a pagoda function is formulated as a *linear programming problem* and solvable in *polynomial* time.
  - With more time, the **pagoda function** may have been useful in determining the outcome of a game earlier, thus reducing the need for playing entire games every time a new individual is created
- In 1999 peg solitaire was completely solved on a computer using an *exhaustive search* through all possible variants. It was achieved making use of the symmetries, efficient storage of board constellations and hashing.
  - Brute force methods are, as expected, totally feasible for peg solitaire. However, they are likely far less efficient than utilizing a GA.
- Shortest solution on a triangular board
  - A solution where the final peg arrives at the **initial empty hole** is *not possible* for a hole in one of the three central positions (corners). An empty corner-hole setup can be solved in **ten moves**.

Resources

- Peg Solitaire - https://en.wikipedia.org/wiki/Peg_solitaire
- Genetic Algorithm - https://en.wikipedia.org/wiki/Genetic_algorithm
- Nils Aall Barricelli - https://en.wikipedia.org/wiki/Nils_Aall_Barricelli
- MarI/O - https://www.youtube.com/watch?v=qv6UVOQ0F44
- ‘NEAT’ Genetic Algorithm - http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf
- ‘Modelling and Solving English Peg Solitaire’ - https://hugues-talbot.github.io/files/Peg_Solitaire_1.pdf
- My Project Specifications - http://pi.cs.oswego.edu/~bdrusche/coursework/csc466/assignments/Project%20Specifications.pdf

Questions or Comments?