

Tower of Hanoi:

By changing the first if condition (test section) to true, we get the basic understanding of what the valid and invalid moves should be and the disc are to be stacked on the poles from big-small as described in the `is_valid_state` function.

Moving from poles i.e. from 0 - 1- 2. Based on the sequence in the moves while also avoiding invalid states and thus adding additional moves while considering that the game should optimally end with 7 moves, I added the code below which also provides the evidence of using $2^n - 1$ to determine optimal steps needed for the game to finish.

```
### 3. Complete the sequence of moves to solve the game
moves = [(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 2)]
for src, dest in moves:
    print('> Moving from', src, 'to', dest)
    s = move_disk(s, src, dest)
    print_poles_as_text(s, n)
    #print(is_valid_state(s))
    print_poles_as_state(s, test_valid=True)
print('Done.')
```

Updating the Random Search was necessary to remove the pointless case of setting src and dest to be the same pole which obviously would have resulted in no state change to overcome this we changed it so that src could be picked from a sample of 0,1,2 while the dest was set to 2.

Using Random Search:

```
src, dest = 0, 0
```

Output of 10 runs

1. Limit = 100, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=100)

if __name__ == "__main__":
    if True:
        towers_of_hanoi

***|***
State: ([],[3, 2, 1]) True
Done.
Guessing with n=3, limit=100 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 100
Moves (valid and kept): 34
Done.
```

2. Limit = 200, No Solution

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=200)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi
            ***|***
            State: ([],[],[3, 2, 1]) True
            Done.
            Guessing with n=3, limit=200 ...
            Result: Hit Limit. No solution.
            Count (attempted random moves) 200
            Moves (valid and kept): 73
            Done.

Process finished with exit code 0
```

3. Limit = 300, No Solution

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=300)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi
            Done.
            Guessing with n=3, limit=300 ...
            Result: Hit Limit. No solution.
            Count (attempted random moves) 300
            Moves (valid and kept): 110
            Done.

Process finished with exit code 0
```

4. Limit = 400, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=400)

### 5 Generate the recursive sequence of moves
if False:
    solve_using_recursion(n=3)

if __name__ == "__main__" > if True
towers_of_hanoi x
Done.
Guessing with n=3, limit=400 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 400
Moves (valid and kept): 400
Done.

Process finished with exit code 0
```

5. Limit = 500, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=500)

### 5 Generate the recursive sequence of moves
if False:
    solve_using_recursion(n=3)

if __name__ == "__main__" > if True
towers_of_hanoi x
Done.
Guessing with n=3, limit=500 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 500
Moves (valid and kept): 500
Done.

Process finished with exit code 0
```

6. Limit = 600, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=600)

### 5 Generate the recursive sequence of moves
if False:
    solve_using_recursion(n=3)

if __name__ == "__main__":
    if True:
        towers_of_hanoi

Done.
Guessing with n=3, limit=600 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 600
Moves (valid and kept): 600
Done.

Process finished with exit code 0
```

7. Limit = 700, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=700)

### 5 Generate the recursive sequence of moves
if False:
    solve_using_recursion(n=3)

if __name__ == "__main__":
    if True:
        towers_of_hanoi

Done.
Guessing with n=3, limit=700 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 700
Moves (valid and kept): 700
Done.

Process finished with exit code 0
```

8. Limit = 800, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=800)

### 5 Generate the recursive sequence of moves
if False:
    solve_using_recursion(n=3)

if __name__ == "__main__":
    towers_of_hanoi
```

Done.
Guessing with n=3, limit=800 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 800
Moves (valid and kept): 800
Done.
Process finished with exit code 0

9. Limit = 900, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=900)

### 5 Generate the recursive sequence of moves
if False:
    solve_using_recursion(n=3)

if __name__ == "__main__":
    towers_of_hanoi
```

Done.
Guessing with n=3, limit=900 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 900
Moves (valid and kept): 900
Done.
Process finished with exit code 0

10. Limit = 1000, No Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=1000)

### 5 Generate the recursive sequence of moves
if False:
    solve_using_recursion(n=3)

if __name__ == "__main__":
    if True:
        towers_of_hanoi

Done.
Guessing with n=3, limit=1000 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 1000
Moves (valid and kept): 1000
Done.

Process finished with exit code 0
```

After many attempts it was made clear that the random search being used was not the best way to achieve the desired outcome, to overcome this we had to remove/avoid the pointless case of when a the same move was being done in reversed and also the loops that if forms by going back and forth in the states.

```
_src, _dest = history[-1]
while True:
    src, dest = sample([0, 1, 2], 2)
    if (src != _dest) or (dest != _src):
        break
```

Output of 10 runs

1. Limit = 100

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=100)

if __name__ == "__main__":
    if True:
        towers_of_hanoi

|      |      ***|***
State: ([],[],[3, 2, 1]) True
Done.
Guessing with n=3, limit=100 ...
Result: Hit Limit. No solution.
Count (attempted random moves) 100
Moves (valid and kept): 34
Done.
```

2. Limit = 200, No Solution

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=200)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi
            ***|***
            State: ([],[],[3, 2, 1]) True
            Done.
            Guessing with n=3, limit=200 ...
            Result: Hit Limit. No solution.
            Count (attempted random moves) 200
            Moves (valid and kept): 73
            Done.

Process finished with exit code 0
```

3. Limit = 300, No Solution

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=300)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi
            Done.
            Guessing with n=3, limit=300 ...
            Result: Hit Limit. No solution.
            Count (attempted random moves) 300
            Moves (valid and kept): 110
            Done.

Process finished with exit code 0
```

4. Limit = 400, Solution found at 325

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=400)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi

Guessing with n=3, limit=400 ...
Result: Found it!
Count (attempted random moves) 325
[(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1),
Moves (valid and kept): 118
Done.

Process finished with exit code 0
```

5. Limit = 500, No new Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=500)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi

Guessing with n=3, limit=500 ...
Result: Found it!
Count (attempted random moves) 325
[(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1),
Moves (valid and kept): 118
Done.

Process finished with exit code 0
```


6. Limit = 600, No new Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=600)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi
            Guessing with n=3, limit=600 ...
            Result: Found it!
            Count (attempted random moves) 325
            [(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1),
            Moves (valid and kept): 118
            Done.

            Process finished with exit code 0
```

7. Limit = 700, No new Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=700)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi
            Guessing with n=3, limit=700 ...
            Result: Found it!
            Count (attempted random moves) 325
            [(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1),
            Moves (valid and kept): 118
            Done.

            Process finished with exit code 0
```

8. Limit = 800, No new Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=800)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True

towers_of_hanoi x
Guessing with n=3, limit=800 ...
Result: Found it!
Count (attempted random moves) 325
[(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1)]
Moves (valid and kept): 118
Done.

Process finished with exit code 0
```

9. Limit = 900, No new Solution found

```
### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=900)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True

towers_of_hanoi x
Guessing with n=3, limit=900 ...
Result: Found it!
Count (attempted random moves) 325
[(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1)]
Moves (valid and kept): 118
Done.

Process finished with exit code 0
```

10. Limit = 1000, No new Solution found

```

### 4 Try to find the solution using random (but valid) guesses for each move
if True:
    attempt_using_random_moves(n=3, limit=1000)

### 5 Generate the recursive sequence of moves
if False:
    if __name__ == "__main__":
        if True:
            towers_of_hanoi

Guessing with n=3, limit=1000 ...
Result: Found it!
Count (attempted random moves) 325
[(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1),
Moves (valid and kept): 118
Done.

Process finished with exit code 0

```

Hence considering the valid moves made and count recorded so far it definitely provides evidence to the reduced number of count and hence shows that the length of sequence to be unaffected in the case of increasing the limit while using better random search.

Output:

```

06 - Lab - Graphs Search Rules towers_of_hanoi.py
towers_of_hanoi.py water_jug_problem.py
Run: towers_of_hanoi
C:\Users\cool\AppData\Local\Programs\Python\Python38-32\python.exe "C:/Users/cool/Documents/AI for games/06 - Lab - Graphs Search Rules/towers_of_hanoi.py"
--poles--
0 : [7, 6, 5, 4, 3, 2, 1]
1 : []
2 : []
State: ([7, 6, 5, 4, 3, 2, 1], [], [])
*|*
**|**
***|***
****|****
*****|*****
*****|*****
*****|*****
True
*|*
**|**
***|***
****|****
*****|*****
*****|*****
*****|*****
True
*|*
**|**
***|***
****|****
*****|*****
*****|*****
*****|*****
False

```

```

06 - Lab - Graphs Search Rules  towers_of_hanoi.py
towers_of_hanoi.py  water_jug_problem.py
Run: towers_of_hanoi
False
***|***
****|****
*****|*****
*****|*****
*****|*****
*****|*****
True
Running coded sequence ...
*|*
**|**
***|***
> Moving from 0 to 2
*|*
**|**
***|***
State: ([3, 2], [], [1]) True
> Moving from 0 to 1
*|*
**|**
***|***
State: ([3], [2], [1]) True
> Moving from 2 to 1
*|*
**|**
***|***
State: ([3], [2, 1], []) True
> Moving from 0 to 2

```

```

06 - Lab - Graphs Search Rules  towers_of_hanoi.py
towers_of_hanoi.py  water_jug_problem.py
Run: towers_of_hanoi
State: ([3], [2, 1], []) True
> Moving from 0 to 2
*|*
**|**
***|***
State: ([], [2, 1], [3]) True
> Moving from 1 to 0
*|*
**|**
***|***
State: ([1], [2], [3]) True
> Moving from 1 to 2
*|*
**|**
***|***
State: ([1], [], [3, 2]) True
> Moving from 0 to 2
*|*
**|**
***|***
State: ([], [], [3, 2, 1]) True
Done.
Guessing with n=3, limit=600 ...
Result: Found it!
Count (attempted random moves) 325
[(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 1), (2, 0), (1, 0), (2, 1), (0, 1), (1, 2), (2, 0), (0, 1), (0, 2), (1, 0), (0, 2), (2, 1), (1, 0), (2, 1)]
Moves (valid and kept): 118
Done.
Process finished with exit code 0

```

Water Jugs Problem:

The water jugs problem involves two jugs where jug 1 can hold up to 5 litres and the second one can hold up to 3 litres.

The game revolves around 3 essential moves to finish namely.

1. Fill a jug
2. Pour a jug into another jug
3. Empty a jug

The way they are represented in the start is something like this (0, 0) == both jugs empty, and for both the jugs to be full it has to be something like this (5, 3)

I have put some comments on the code to make it more understandable while fixing the assert test, hope it helps

Test Filling:

```
if __name__ == "__main__":
    ### 1 Basic testing of methods and operations
    if True:
        JUG_CFG = [5,3] # (Die Hard movie version)
        s = setup_jugs()
        # both jugs empty
        #print("Im here 0,0 both jugs empty")
        print(s)
        # test fillling
        print("Test Filling")
        s = fill(s, 0)
        #print("Im here 5,0 pour jug1 = 5 and jug2= 0, pour 1 into 2")
        print(s)
        assert s == (5, 0)
        s = fill(s, 1)
        #print("Im here 5,3 jug1=5 while jug2 = 3, both jugs full ")
        print(s)
        assert s == (5, 3)
        #print("Im here both jugs full")
        print(s)
```

Test Empty:

```
# test empty
print("Test Empty")
s = empty(s, 0)
#print("Im here 0,3 jug1=0 while jug2 = 3, now transfer into jug 1 ")
print(s)
assert s == (0, 3)
s = empty(s, 1)
#print("Im here 0,0 jug1=0 while jug2 = 0, both jugs empty")
print(s)
assert s == (0, 0)
```


Test Pour:

```
# test pour / leftover actions
print("testing pour")
s = fill(s, 0)
assert s == (5, 0)
s = pour(s, 0, 1)
#print("Im here jug1= 2 and jug2= 3, empty 2")
print(s)
assert s == (2, 3)
s = empty(s, 0)
#print("Im here jug1= 0 and jug2= 3 ,transfer to jug1")
print(s)
assert s == (0, 3)
s = pour(s, 1, 0)
#print("Im here jug1= 3 and jug2= 0 ,fill jug2")
print(s)
assert s == (3, 0)
s = fill(s, 1)
#print("Im here jug1=3 and jug2= 3 ,fill jug 1 from jug 2")
print(s)
assert s == (3, 3)
s = pour(s, 1, 0)
#print("Im here jug1= 5 and jug2= 1 ,empty jug 1")
print(s)
assert s == (5, 1)
```

```
C:\Users\cools\AppData\Local\Programs\Python\Python38-32\python.exe "C:/Users/
(0, 0)
Test Filling
(5, 0)
(5, 3)
(5, 3)
Test Empty
(0, 3)
(0, 0)
testing pour
(2, 3)
(0, 3)
(3, 0)
(3, 3)
(5, 1)
```

Completing Sequence 1:

Here filling in the sequence to achieve result (4,0) == jug 1 holds around 4 litres while the jug 2 is empty. In order to get these results the from the initial stage of being provided with fill which takes one parameter and fills up the first jug then following the pour which takes up two arguments, (0, 1) where it pours into jug 2 which leaves jug 1 with only 2 litres remaining, followed by empty which takes in a single argument to empty the second jug and thus it follows the sequence to achieve the result, see below for complete sequence....

```

### 2 Solve using a pre-defined sequence of actions
if True:
    action_calls = {
        'fill': fill,
        'empty': empty,
        'pour': pour
    }
    print('Doing sequence 1 ...')
    ### 3 Sequence 1 of moves
    actions = [
        # tuples, string of method to call then arguments to call
        ('fill', (0,)),
        ('pour', (0, 1)), # (5,0) pour 1 into 2
        ('empty', (1,)), # (2,3) empty 2
        ('pour', (0, 1)), # (2,0) transfer from 1 to 2
        ('fill', (0,)), # (0,2) fill jug 1
        ('pour', (0, 1)), # (5,2) fill jug 2 from 1
        ('empty', (1,)), # (4,3) discarded jug 2

        # result should be (4,0)
    ]

```

```

Doing sequence 1 ...
Calling... fill with (0,) on (0, 0)
(5, 0)
Calling... pour with (0, 1) on (5, 0)
(2, 3)
Calling... empty with (1,) on (2, 3)
(2, 0)
Calling... pour with (0, 1) on (2, 0)
(0, 2)
Calling... fill with (0,) on (0, 2)
(5, 2)
Calling... pour with (0, 1) on (5, 2)
(4, 3)
Calling... empty with (1,) on (4, 3)
(4, 0)
Done

```

Completing Sequence 2:

Here filling in the sequence to achieve result (4,0) == jug 1 holds around 4 litres while the jug 2 is empty. In order to get these results the from the initial stage of being provided with fill which takes one parameter and fills up the second jug then following the pour which takes up two arguments, (1, 0) where it pours into jug 1 which leaves jug 2 empty, followed by fill which takes in a single argument to fill the second jug and thus it follows the sequence to achieve the result, see below for complete sequence....

```
### 4 Solve using sequence 2
if True:
    action_calls = {
        'fill': fill,
        'empty': empty,
        'pour': pour
    }

    print('Doing sequence 2 ...')
    actions = [
        # tuples, string of method to call then arguments to call
        ('fill', [1]),      # fill jug 2 => (0,3)
        ('pour', [1, 0]),   # transfer 2 to jug 1 => (3,0)
        ('fill', [1]),      # fill jug 2 => (3,3)
        ('pour', [1, 0]),   # fill jug 1 into jug 2 => (5,1)
        ('empty', [0]),     # empty jug 1  => (0,1)
        ('pour', [1, 0]),   # transfer from jug 2 to jug 1  => (1,0)
        ('fill', [1]),      # fill jug 2  => (1,3)
        ('pour', [1, 0]),   # transfer from jug 2 to jug 1  => (4,0)

        # result should be (4,0)
    ]

    # run sequence of actions
```

```
Doing sequence 2 ...
Calling... fill with [1] on (0, 0)
(0, 3)
Calling... pour with [1, 0] on (0, 3)
(3, 0)
Calling... fill with [1] on (3, 0)
(3, 3)
Calling... pour with [1, 0] on (3, 3)
(5, 1)
Calling... empty with [0] on (5, 1)
(0, 1)
Calling... pour with [1, 0] on (0, 1)
(1, 0)
Calling... fill with [1] on (1, 0)
(1, 3)
Calling... pour with [1, 0] on (1, 3)
(4, 0)
Done
```

Solving using Random Search:

Considering all possible moves with only 8 unique moves for a two jug problem, but if we were to exclude the pointless action of pouring the same jug over and over again it would only leave us with six (pour from 1 to 1 and pour from 2) actions.

All possible actions excluding the pointless actions;

```
# Random choice from all possible actions for a fixed problem
if True:
    # There is a set of six unique actions to choose from
    actions = [
        # all possible fill's
        (fill, [0]),
        (fill, [1]),
        # all possible pour's
        (pour, [0, 1]),
        (pour, [1, 0]),
        # all possible empty's
        (empty, [0]),
        (empty, [1]),
    ]
    # Notes:
    # - We exclude pour 0->0 and 1->1 as they are pointless
    # - Some actions might have no effect (empty if already empty)
    #   but we are not making conditional actions (only naive ones)
```

Passing the arguments

```
# select a random action to try
fn, args = choice(actions)
new_s = fn(s, *args)
```

```
if new_s:
    history.append((fn, args))
    if new_s == s_end:
        status = 'Success'
```

The above condition keeps the the outcome if it is valid (Not NONE!), while the choice function returns a random item from the list of actions passed to it.

```
Trying a random action search:
(0, 0) (0, 0) (0, 0) (0, 3) (0, 0) (0, 3) (0, 0) (0, 0) (5, 0) (5, 0) (0, 0) (0, 3) (0, 3) (0, 0) (0, 0) (0, 0) (0, 3) (0, 0) (0, 0) (0, 0) (0, 3) (5, 0)
Result: Hit limit (limit=4000, count=4000, history=4000)

Process finished with exit code 0
```

It hits limit = 4000 while the History= 4000 and increasing the limit size to 6000 didn't produce much of a change either, also while checking for goal states it couldn't find any even though the limit increased but no solution found

```
Trying a random action search:
(0, 0) (0, 3) (5, 0) (0, 0) (0, 0) (5, 0) (0, 0) (0, 3) (0, 3) (5, 0) (0, 0) (0, 0) (5, 0) (0, 3) (0, 0) (0, 0) (5, 0) (0, 0) (5, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
Result: Hit limit (limit=6000, count=6000, history=6000)

Process finished with exit code 0
```


Trying a better Random Search:

By adding more valid states where the second jug can have up to it's max water level as we are only targeting the first jug as 4 and the second one is of no concern to us and by putting it in the list it creates more valid end states to check against.

```
###TODO: use a list of valid end states, not just one
end_states = [(4, 0), (4, 1), (4, 2), (4, 3)]
```

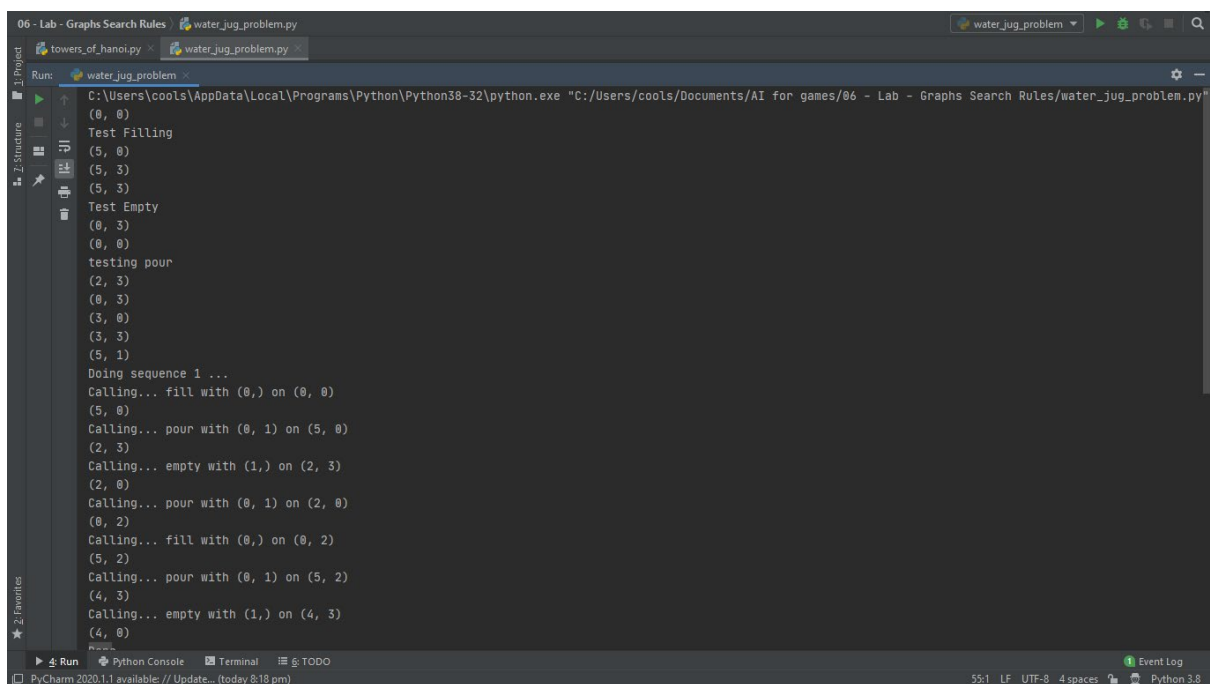
```
# if move outcome state is valid (not None) keep it
if new_s and new_s in s:
    s = new_s
    history.append((fn, args))
    if new_s in s_end:
        status = 'Success'
```

By using the above code it can be seen that the history has been reduced to zero even though the guess count stays the same which mean that the same number of the guess were used for the sequence of states in the history to be smaller to succeed.

```
Trying a random action search:
(0, 0) (0, 0) (5, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (0, 3) (0, 0) (0, 0) (5, 0) (0, 0) (0, 0) (0, 0) (0, 3) (0, 0) (0, 0) (5, 0) (0, 0) (0, 0) (0, 3)
Result: Hit limit (limit=4000, count=4000, history=0)

Process finished with exit code 0
```

Output:



```
06 - Lab - Graphs Search Rules | water_jug_problem.py
towers_of_hanoi.py | water_jug_problem.py
Run: water_jug_problem
C:\Users\cools\AppData\Local\Programs\Python\Python38-32\python.exe "C:/Users/cools/Documents/AI for games/06 - Lab - Graphs Search Rules/water_jug_problem.py"
(0, 0)
Test Filling
(5, 0)
(5, 3)
(5, 3)
Test Empty
(0, 3)
(0, 0)
testing pour
(2, 3)
(0, 3)
(3, 0)
(3, 3)
(5, 1)
Doing sequence 1 ...
Calling... fill with (0,) on (0, 0)
(5, 0)
Calling... pour with (0, 1) on (5, 0)
(2, 3)
Calling... empty with (1,) on (2, 3)
(2, 0)
Calling... pour with (0, 1) on (2, 0)
(0, 2)
Calling... fill with (0,) on (0, 2)
(5, 2)
Calling... pour with (0, 1) on (5, 2)
(4, 3)
Calling... empty with (1,) on (4, 3)
(4, 0)
Result: Hit limit (limit=4000, count=4000, history=0)
Process finished with exit code 0
```

```
06 - Lab - Graphs Search Rules | water_jug_problem.py | water_jug_problem
water_jug_problem
towers_of_hanoi.py | water_jug_problem.py
Run: water_jug_problem
(5, 2)
Calling... pour with (0, 1) on (5, 2)
(4, 3)
Calling... empty with (1,) on (4, 3)
(4, 0)
Done
Doing sequence 2 ...
Calling... fill with [1] on (0, 0)
(0, 3)
Calling... pour with [1, 0] on (0, 3)
(3, 0)
Calling... fill with [1] on (3, 0)
(3, 3)
Calling... pour with [1, 0] on (3, 3)
(5, 1)
Calling... empty with [0] on (5, 1)
(0, 1)
Calling... pour with [1, 0] on (0, 1)
(1, 0)
Calling... fill with [1] on (1, 0)
(1, 3)
Calling... pour with [1, 0] on (1, 3)
(4, 0)
Done
Trying a random action search:
(0, 0) (0, 3) (0, 3) (0, 0) (0, 0) (5, 0) (0, 3) (0, 3) (0, 0) (0, 0) (0, 0) (0, 3) (0, 0) (0, 0) (0, 0) (0, 0) (0, 0) (5, 0) (0, 0) (5, 0) (5, 0) (0, 0)
Result: Hit limit (limit=4000, count=4000, history=0)
Process finished with exit code 0
Run Python Console Terminal TODO
PyCharm 2020.1.1 available: // Update... (today 8:18 pm) 55:1 LF UTF-8 4 spaces Python 3.8
```