Solving the Water Jug Puzzle in CLIPS

Feng-Jen Yang, *Member, IAENG*

***Abstract*—Unlike object-oriented programming approaches in which a solution is clearly represented by objects, classes, inheritance and polymorphism, the logic programming approach is focus on the inference that applies the rules in the knowledge-base to enhance the contents in the fact-base until the goal is reached. In this paper, a logical problem solving is illustrated by using CLIPS programming language to solve the water jug puzzle.**

***Index Terms*—State Space, Search Tree, Logical Programming, Forward Chaining Inferencing.**

I. Introduction

In terms of logical programming, most of the real-life problem solving can be achieved by representing a solution as a series of state transitions. After finding a suitable problem state representation, a problem solver can go on to think of all possible operations that can cause a state transition from a given state to its successor states. In this manner, the entire problem domain can be viewed as a state space, also known as a search space, that consists of all possible states and all possible transitions among states. Within this state space, any path that starts from the initial state to a goal state is representing a solution. The series of operations that are performed along the path are the steps of solving this problem. A potential problem with state space while searching for paths is that the directional state transitions might incur loops involved and end up with being trapped by infinite length path. As a result, along with the design of knowledge rules toward problem solving, any operation that will cause a transition from the current state to a previously visited state should be excluded from the search space. In this paper, this problem-solving approach is illustrated to solve the water jug puzzle in CLIPS programming language.

II. The Water Jug Puzzle

The origin of water jug puzzle dates back to mediaeval times when people were playing a mathematical game by using a fixed number of water jugs that can hold different integral units of water volumes but with no measuring marks on them, by filling up jugs from other jugs or emptying jugs into other jugs in a proper sequence, they found that they could get the exact amount of water volume they expected in advance [1]. This kind of puzzle is a good illustration of problem solving by searching the state space to find a sequence of actions that can lead to a sequence of state transitions from the initial state to a goal state.

An instance of the water jug puzzle, which is chosen to be demonstrated and solved in this paper, starts from two given water jugs in which one can hold up to 4 gallons of water and the other can hold up to 3 gallons of water. The game rules are allowing a player to fill up a jug from the tap, empty a jug into the sink, fill up a jug from the other jug, or empty a jug into the other jug. Given these conditions and restrictions, eventually the player is required to fill the 4-gallon just with exactly 2 gallons of water.

Manuscript received July 6, 2018; revised July 27, 2018.

F. Yang is with the Department of Computer Science, Florida Polytechnic University, FL 33805, USA, e-mail: fyang@floridapoly.edu.

A. The Problem State Representation

Since the essential criterion for solving the Water Jug Puzzle is based how much water are filled in each jug, a very nature and intuitive way to represent the problem state is using an ordered pair of numbers to indicate the amount of water in the 4-gallon jug and the 3-gallon jug. So that the ordered pair (x, y) indicates that there is x gallons of water in the 4-gallon jug and y gallons of water in the 3-gallon jug. With this analogy, this problem domain can be treated as search tree in which the initial state is (0, 0) and the goal state is (2, y) where y can be any amount of water left in the 3-gallon jug.

B. The State Transitions

The sate transitions can be derived based on the game rules. With the 4 rules of water pouring and filling, the following 8 possible operations can be derived as follows:

1) Fill up the 4-gallon jug from the tap, i.e., (x, y) $\rightarrow$ (4, y) where x $<$ 4.
2) Fill up the 3-gallon jug from the tap, i.e., (x, y) $\rightarrow$ (x, 3) where y $<$ 3.
3) Fill up the 4-gallon jug from the 3-gallon jug, i.e., (x, y) $\rightarrow$ (4, x+y-4) where x $>$ 0 and x+y $\geq$ 4.
4) Fill up the 3-gallon jug from the 4-gallon jug, i.e., (x, y) $\rightarrow$ (x+y-3, 3) where y $>$ 0 and x+y $\geq$ 3.
5) Empty the 4-gallon jug into the 3-gallon jug, i.e., (x, y) $\rightarrow$ (0, x+y) where x $>$ 0 and x+y $\leq$ 3.
6) Empty the 3-gallon jug into the 4-gallon jug, i.e. (x, y) $\rightarrow$ (x+y, 0) where y $>$ 0 and x+y $\leq$ 4.
7) Empty the 4-gallon jug into the sink, i.e., (x, y) $\rightarrow$ (0, y) where x $>$ 0.
8) Empty the 3-gallon jug into the sin, i.e., (x, y) $\rightarrow$ (x, 0) where y $>$ 0.

The operations 1 and 2 are derived from the 2 possible situations of filling up a jug. The operations 3 and 4 are derived from the 2 possible situations of frilling up a jug from the other jug. The operations 5 and 6 are derived from the 2 possible situations of emptying a jug into the other jug. The operations 7 and 4 are derived from the 2 possible situations of emptying a jug into the sink.

III. Solving the Puzzle in CLIPS Programming Language

While most of the AI literatures are solving the water jugs puzzle by performing a depth first search or breadth first search on a pre-constructed search tree, I am looking at the problem from a different prospective and attempting

ISBN: 978-988-14049-0-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

to solve this problem by using the built-in forward chaining inference engine within the CLIPS programing language. The rationale of exploring this new approach is that we do not have to construct the search tree in advance and then adopting a search technique to look for solutions. Instead, a path from the initial state to a goal state can be both inferred and saved at the same time. The programming details are described in the subsequent sections by using CLIPS syntax [2].

A. Asserting the initial state

The program starts from asserting the initial state to the fact-base which can be done in the following statement:

```
(deffacts the-initial-state
(state 0 0))
```

B. Inferring New States and Links by Applying the Eight Operations

Based on the initial state, we can go on to encode the aforementioned eight operations into the following correspondent knowledge rules:

```
(defrule op1
  (state ?x  ?y)
  (test (< ?x 4))
  (not (exists (state 4 ?y)))
  =>
  (assert (state 4 ?y))
  (assert (link  ?x ?y to 4 ?y)))

(defrule op2
  (state ?x  ?y)
  (test (< ?y 3))
  (not (exists (state ?x 3)))
  =>
  (assert (state ?x 3))
  (assert (link  ?x ?y to ?x 3)))

(defrule op3
  (state ?x ?y)
  (test (> ?x 0))
  (test (> (+ ?x ?y) 4))
  (not (exists (state 4 =(- (+ ?x ?y) 4)
  )))
  =>
  (assert (state 4 =(- (+ ?x ?y) 4)))
  (assert (link ?x ?y to 4 =(- (+ ?x ?y)
   4))))

(defrule op4
  (state ?x ?y)
  (test (> ?y 0))
  (test (> (+ ?x ?y) 3))
  (not (exists (state =(- (+ ?x ?y) 3) 3)
  ))
  =>
  (assert (state =(- (+ ?x ?y) 3) 3))
  (assert (link ?x ?y to =(- (+ ?x ?y) 3)
   3)))

(defrule op5
  (state ?x ?y)
  (test (> ?x 0))
  (test (< (+ ?x ?y) 3))
  (not (exists (state 0 =(+ ?x ?y))))
  =>
  (assert (state 0 =(+ ?x ?y)))
  (assert (link ?x ?y to 0 =(+ ?x ?y))))

(defrule op6
  (state ?x ?y)
  (test (> ?y 0))
  (test (< (+ ?x ?y) 4))
  (not (exists (state =(+ ?x ?y) 0)))
  =>
  (assert (state =(+ ?x ?y) 0))
  (assert (link ?x ?y to =(+ ?x ?y) 0)))

(defrule op7
  (state ?x ?y)
  (test (> ?x 0))
  (not (exists (state 0 ?y)))
  =>
  (assert (state 0 ?y))
  (assert (link ?x ?y to 0 ?y)))

(defrule op8
  (state ?x ?y)
  (test (> ?y 0))
  (not (exists (state ?x 0)))
  =>
  (assert (state ?x 0))
  (assert (link ?x ?y to ?x 0)))
```

C. The Resultant Search Tree

By performing the forward chaining inference and start the initial state, the follow search space is added into the fact-base of CLIPS:

```
(state 0 0), (state 4 0), (state 0 3),
(state 4 3), (state 1 3), (state 3 0),
(state 1 0), (state 3 3), (state 0 1),
(state 4 2), (state 4 1), (state 0 2),
(state 2 3), (state 2 0),
(link (state 0 0) (state 4 0)),
(link (state 0 0) (state 0 3)),
(link (state 4 0) (state 4 3)),
(link (state 4 0) (state 1 3)),
(link (state 0 3) (state 3 0)),
(link (state 1 3) (state 1 0)),
(link (state 3 0) (state 3 3)),
(link (state 1 0) (state 0 1)),
(link (state 3 3) (state 4 2)),
(link (state 0 1) (state 4 1)),
(link (state 4 2) (state 0 2)),
(link (state 4 1) (state 2 3)),
(link (state 0 2) (state 2 0))
```

D. Inferring paths within the Search Tree

After the search space is constructed, the following two knowledge rules can then be used to construct all paths leading from the initial:

```
defrule direct-path
    (link ?x1 ?y1 to ?x2 ?y2)
```

ISBN: 978-988-14049-0-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
    =>
    (assert (path ?x1 ?y1 to ?x2 ?y2
    (str-cat "(" ?x1 ", " ?y1 ") -->
    (" ?x2 ", " ?y2 ")"))))

(defrule indirect-path
    (path ?x1 ?y1 to ?x2 ?y2 ?route)
    (link ?x2 ?y2 to ?x3 ?y3)
    =>
    (assert (path ?x1 ?y1 to ?x3 ?y3
    (str-cat ?route " --> (" ?x3 ", "
    ?y3 ")"))))
```

E. The Resultant Paths

After the inference of all paths starting from the initial state, the following paths are added to the fact-base of CLIPS:

```
(path 4 3 to 2 3 "(4, 3) --> (0, 3) -->
(3, 0) --> (3, 3) --> (4, 2) --> (0, 2)
--> (2, 0) --> (2, 3)")

(path 4 0 to 2 3 "(4, 0) --> (4, 3) -->
(0, 3) --> (3, 0) --> (3, 3) --> (4, 2)
--> (0, 2) --> (2, 0) --> (2, 3)")

(path 0 0 to 2 3 "(0, 0) --> (4, 0) -->
(4, 3) --> (0, 3) --> (3, 0) --> (3, 3)
--> (4, 2) --> (0, 2) --> (2, 0) -->
(2, 3)")

(path 2 3 to 4 1 "(2, 3) --> (4, 1)")

(path 2 0 to 4 1 "(2, 0) --> (2, 3) -->
(4, 1)")

(path 0 2 to 4 1 "(0, 2) --> (2, 0) -->
(2, 3) --> (4, 1)")

(path 4 2 to 4 1 "(4, 2) --> (0, 2) -->
(2, 0) --> (2, 3) --> (4, 1)")

(path 3 3 to 4 1 "(3, 3) --> (4, 2) -->
(0, 2) --> (2, 0) --> (2, 3) --> (4, 1)")

(path 3 0 to 4 1 "(3, 0) --> (3, 3) -->
(4, 2) --> (0, 2) --> (2, 0) --> (2, 3)
--> (4, 1)")

(path 0 3 to 4 1 "(0, 3) --> (3, 0) -->
(3, 3) --> (4, 2) --> (0, 2) --> (2, 0)
--> (2, 3) --> (4, 1)")

(path 4 3 to 4 1 "(4, 3) --> (0, 3) -->
(3, 0) --> (3, 3) --> (4, 2) --> (0, 2)
--> (2, 0) --> (2, 3) --> (4, 1)")

(path 4 0 to 4 1 "(4, 0) --> (4, 3) -->
(0, 3) --> (3, 0) --> (3, 3) --> (4, 2)
--> (0, 2) --> (2, 0) --> (2, 3) -->
(4, 1)")

(path 0 0 to 4 1 "(0, 0) --> (4, 0) -->
(4, 3) --> (0, 3) --> (3, 0) --> (3, 3)
--> (4, 2) --> (0, 2) --> (2, 0) -->
(2, 3) --> (4, 1)")

(path 4 1 to 0 1 "(4, 1) --> (0, 1)")

(path 2 3 to 0 1 "(2, 3) --> (4, 1) -->
(0, 1)")

(path 2 0 to 0 1 "(2, 0) --> (2, 3) -->
(4, 1) --> (0, 1)")

(path 0 2 to 0 1 "(0, 2) --> (2, 0) -->
(2, 3) --> (4, 1) --> (0, 1)")

(path 4 2 to 0 1 "(4, 2) --> (0, 2) -->
(2, 0) --> (2, 3) --> (4, 1) --> (0, 1)")

(path 3 3 to 0 1 "(3, 3) --> (4, 2) -->
(0, 2) --> (2, 0) --> (2, 3) --> (4, 1)
--> (0, 1)")

(path 3 0 to 0 1 "(3, 0) --> (3, 3) -->
(4, 2) --> (0, 2) --> (2, 0) --> (2, 3)
--> (4, 1) --> (0, 1)")

(path 0 3 to 0 1 "(0, 3) --> (3, 0) -->
(3, 3) --> (4, 2) --> (0, 2) --> (2, 0)
--> (2, 3) --> (4, 1) --> (0, 1)")

(path 4 3 to 0 1 "(4, 3) --> (0, 3) -->
(3, 0) --> (3, 3) --> (4, 2) --> (0, 2)
--> (2, 0) --> (2, 3) --> (4, 1) -->
(0, 1)")

(path 4 0 to 0 1 "(4, 0) --> (4, 3) -->
(0, 3) --> (3, 0) --> (3, 3) --> (4, 2)
--> (0, 2) --> (2, 0) --> (2, 3) -->
(4, 1) --> (0, 1)")

(path 0 0 to 0 1 "(0, 0) --> (4, 0) -->
(4, 3) --> (0, 3) --> (3, 0) --> (3, 3)
--> (4, 2) --> (0, 2) --> (2, 0) -->
(2, 3) --> (4, 1) --> (0, 1)")

(path 0 1 to 1 0 "(0, 1) --> (1, 0)")

(path 4 1 to 1 0 "(4, 1) --> (0, 1) -->
(1, 0)")

(path 2 3 to 1 0 "(2, 3) --> (4, 1) -->
(0, 1) --> (1, 0)")

(path 2 0 to 1 0 "(2, 0) --> (2, 3) -->
(4, 1) --> (0, 1) --> (1, 0)")

(path 0 2 to 1 0 "(0, 2) --> (2, 0) -->
(2, 3) --> (4, 1) --> (0, 1) --> (1, 0)")
```

ISBN: 978-988-14049-0-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

```
(path 4 2 to 1 0 "(4, 2) --> (0, 2) -->
(2, 0) --> (2, 3) --> (4, 1) --> (0, 1)
--> (1, 0)")

(path 3 3 to 1 0 "(3, 3) --> (4, 2) -->
(0, 2) --> (2, 0) --> (2, 3) --> (4, 1)
--> (0, 1) --> (1, 0)")

(path 3 0 to 1 0 "(3, 0) --> (3, 3) -->
(4, 2) --> (0, 2) --> (2, 0) --> (2, 3)
--> (4, 1) --> (0, 1) --> (1, 0)")

(path 0 3 to 1 0 "(0, 3) --> (3, 0) -->
(3, 3) --> (4, 2) --> (0, 2) --> (2, 0)
--> (2, 3) --> (4, 1) --> (0, 1) -->
(1, 0)")

(path 4 3 to 1 0 "(4, 3) --> (0, 3) -->
(3, 0) --> (3, 3) --> (4, 2) --> (0, 2)
--> (2, 0) --> (2, 3) --> (4, 1) -->
(0, 1) --> (1, 0)")

(path 4 0 to 1 0 "(4, 0) --> (4, 3) -->
(0, 3) --> (3, 0) --> (3, 3) --> (4, 2)
--> (0, 2) --> (2, 0) --> (2, 3) -->
(4, 1) --> (0, 1) --> (1, 0)")

(path 0 0 to 1 0 "(0, 0) --> (4, 0) -->
(4, 3) --> (0, 3) --> (3, 0) --> (3, 3)
--> (4, 2) --> (0, 2) --> (2, 0) -->
(2, 3) --> (4, 1) --> (0, 1) --> (1, 0)")

(path 1 0 to 1 3 "(1, 0) --> (1, 3)")

(path 0 1 to 1 3 "(0, 1) --> (1, 0) -->
(1, 3)")

(path 4 1 to 1 3 "(4, 1) --> (0, 1) -->
(1, 0) --> (1, 3)")

(path 2 3 to 1 3 "(2, 3) --> (4, 1) -->
(0, 1) --> (1, 0) --> (1, 3)")

(path 2 0 to 1 3 "(2, 0) --> (2, 3) -->
(4, 1) --> (0, 1) --> (1, 0) --> (1, 3)")

(path 0 2 to 1 3 "(0, 2) --> (2, 0) -->
(2, 3) --> (4, 1) --> (0, 1) --> (1, 0)
--> (1, 3)")

(path 4 2 to 1 3 "(4, 2) --> (0, 2) -->
(2, 0) --> (2, 3) --> (4, 1) --> (0, 1)
--> (1, 0) --> (1, 3)")

(path 3 3 to 1 3 "(3, 3) --> (4, 2) -->
(0, 2) --> (2, 0) --> (2, 3) --> (4, 1)
--> (0, 1) --> (1, 0) --> (1, 3)")

(path 3 0 to 1 3 "(3, 0) --> (3, 3) -->
(4, 2) --> (0, 2) --> (2, 0) --> (2, 3)
--> (4, 1) --> (0, 1) --> (1, 0) -->
(1, 3)")

(path 0 3 to 1 3 "(0, 3) --> (3, 0) -->
(3, 3) --> (4, 2) --> (0, 2) --> (2, 0)
--> (2, 3) --> (4, 1) --> (0, 1) -->
(1, 0) --> (1, 3)")

(path 4 3 to 1 3 "(4, 3) --> (0, 3) -->
(3, 0) --> (3, 3) --> (4, 2) --> (0, 2)
--> (2, 0) --> (2, 3) --> (4, 1) -->
(0, 1) --> (1, 0) --> (1, 3)")

(path 4 0 to 1 3 "(4, 0) --> (4, 3) -->
(0, 3) --> (3, 0) --> (3, 3) --> (4, 2)
--> (0, 2) --> (2, 0) --> (2, 3) -->
(4, 1) --> (0, 1) --> (1, 0) --> (1, 3)")

(path 0 0 to 1 3 "(0, 0) --> (4, 0) -->
(4, 3) --> (0, 3) --> (3, 0) --> (3, 3)
--> (4, 2) --> (0, 2) --> (2, 0) -->
(2, 3) --> (4, 1) --> (0, 1) --> (1, 0)
--> (1, 3)")
```

F. Printing successfully solutions

Finally, by using the following knowledge rule we can display solutions in the format of paths starting from the initial state and ending at a goal state:

(defrule print-solutions (path 0 0 to 2 ?y ?route) =¿ (printout t ?route crlf))

G. The Final Solutions

Eventually, the solutions are displayed as follows:

```
(0, 0) --> (4, 0) --> (4, 3) --> (0, 3)
--> (3, 0) --> (3, 3) --> (4, 2) -->
(0, 2) --> (2, 0)

(0, 0) --> (4, 0) --> (4, 3) --> (0, 3)
--> (3, 0) --> (3, 3) --> (4, 2) -->
(0, 2) --> (2, 0) --> (2, 3)
```

IV. Conclusion

Even though most of the college students are more familiar and comfortable with imperative and object-oriented paradigms of programming, logical programing paradigm is more suitable of solving AI problems. In this paper, a sample program in CLIPS programming language is illustrated to solve the water jug puzzle. Instead of aiming at inventing new theory or problem-solving method, this illustration is for the purpose of providing an additional problem-solving example to AI related studies.

References

[1] E. B. Cowley, ”Note on a Linear Diophantine Equation, Questions and Discussions,” *American Mathematical Monthly*, Vol. 33, No. 7, pp. 379381, 1926.

[2] J. C. Giarratano, ”CLIPS Users Guide,” Version 6.30, *http://clipsrules.sourceforge.net/OnlineDocs.html*, 2015.

ISBN: 978-988-14049-0-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)