AI: “Water Jug Problem” logical concept

Shaikh Maqsood, Ansari Ammar, Mishra Manu, Bhor Viraj,
Students TE IT KCCMSR

Abstract:
Artificial intelligence as the word itself defines the worst domain, consisting of future aspect which focuses on mankind. Let’s concentrate on one simple aspect to build a view of logic in us to support Intelligence in machine or rather Artificial Intelligence in m/c in every aspect. But we all mainly concern for goal oriented intelligence rather than Human kind intelligence. Just take one logical base problem “Water Jug Problem”. In this paper we will see the solution of water jug problem in Prolog and Problem definition.
General Terms: Problem solving, water jug problem, problem finding, problem query, intelligence.

Keyword: Artificial Intelligence, Problem Solving Technique, Logics , Prolog.

1. Introduction: The "Water-jugs Problem"

This classic AI problem is described in Artificial Intelligence as follows:
"You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?".
This program implements an "environmentally responsible" solution to the water jugs problem. Rather than filling and spilling from an infinite water resource, we conserve a finite initial charge with a third jug: (reservoir).
This approach is simpler than the traditional method, because there are only two actions; it is more flexible than the traditional method, because it can solve problems that are constrained by a limited supply from the reservoir.
To simulate the infinite version, we use a filled reservoir with a capacity greater than the combined capacities of the jugs, so that the reservoir can never be emptied.
"Perfection is achieved not when there is nothing more to add, but when there is nothing more to take away." Antoine de Saint-Exupéry”

2. Problem Solving:

A problem is an obstacle, impediment, difficulty or challenge, or any situation that invites resolution; the resolution of which is recognized as a solution or contribution toward a known purpose or goal. A problem implies a desired outcome coupled with an apparent deficiency, doubt or inconsistency that prevents the outcome from taking place.

Every theoretical problem asks for an answer or solution. Trying to find a solution to a problem is known as problem solving. There are many standard techniques for problem solving, such as Proof by Contradiction, or Proof by Exhaustion, the latter famously being used in the solution to the Thirty-Six Officers Problem posed by Leonhard Euler. A problem is a gap between an actual and desired situation. The time it takes to solve a problem is a way of measuring complexity.[1] Many problems have no discovered solution and are therefore classified as an open problem.

Problem finding means problem discovery. It is part of the larger problem process that includes problem shaping and problem solving. Problem finding requires intellectual vision and insight into what is missing. This involves the application of creativity.

Finding a problem can, depending on the problem, be either much easier or much harder than solving the problem. An example of a problem that was much easier to find than to solve is Fermat's Last Theorem. The problem is simple, is it true that it is impossible to separate any power higher than the second into two like powers? Solving the problem took 357 years

Problem solving is a mental process which is part of the larger problem process that includes problem finding and problem shaping. Considered the most complex of all intellectual functions, problem solving has been defined as higher-order cognitive process that requires the modulation and control of more routine or fundamental skills.[1] Problem solving occurs when an organism or an artificial intelligence system needs to move from a given state to a desired goal state.

3. Define the problem

This is often where people struggle. They react to what they think the problem is. Instead, seek to understand more about why you think there's a problem.

Define the problem: (with input from yourself and others). Ask yourself and others, the following questions:

1. What can you see that causes you to think there's a problem?
2. Where is it happening?
3. How is it happening?
4. When is it happening?
5. With whom is it happening? (HINT: Don't jump to "Who is causing the problem?" When we're stressed, blaming is often one of our first reactions. To be an effective manager, you need to address issues more than people.)
6. Why is it happening?
7. Write down a five-sentence description of the problem in terms of "The following should be happening, but isn't ..." or "The following is happening and should be: ..." As much as possible, be specific in your description, including what is happening, where, how, with whom and why. (It may be helpful at this point to use a variety of research methods.

Defining complex problems:

If the problem still seems overwhelming, break it down by repeating steps 1-7 until you have descriptions of several related problems.

Verifying your understanding of the problems:

It helps a great deal to verify your problem analysis for conferring with a peer or someone else.

Prioritize the problems:

If you discover that you are looking at several related problems, then prioritize which ones you should address first.
Note the difference between "important" and "urgent" problems. Often, what we consider to be important problems to consider are really just urgent problems. Important problems deserve more attention. For example, if you're continually answering "urgent" phone calls, then you've probably got a more "important" problem and that's to design a system that screens and prioritizes your phone calls.

Understand your role in the problem:

Your role in the problem can greatly influence how you perceive the role of others. For example, if you're very stressed out, it'll probably look like others are, too, or, you may resort too quickly to blaming and reprimanding others. Or, you are feel very guilty about your role in the problem, you may ignore the accountabilities of others.

2. Look at potential causes for the problem

- It's amazing how much you don't know about what you don't know. Therefore, in this phase, it's critical to get input from other people who notice the problem and who are effected by it.
- It's often useful to collect input from other individuals one at a time (at least at first). Otherwise, people tend to be inhibited about offering their impressions of the real causes of problems.
- Write down what your opinions and what you've heard from others.
- Regarding what you think might be performance problems associated with an employee, it's often useful to seek advice from a peer or your supervisor in order to verify your impression of the problem.
- Write down a description of the cause of the problem and in terms of what is happening, where, when, how, with whom and why.

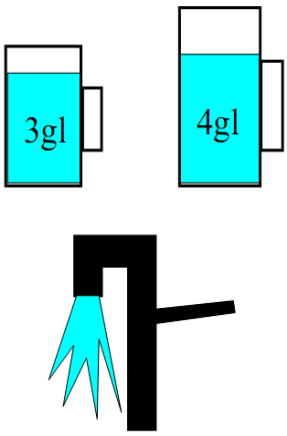
3. Identify alternatives for approaches to resolve the problem

At this point, it's useful to keep others involved (unless you're facing a personal and/or employee performance problem). Brainstorm for solutions to the problem. Very simply put, brainstorming is collecting as many ideas as possible, then screening them to find the best idea. It's critical when collecting the ideas to not pass any judgment on the ideas -- just write them down as you hear them. (A wonderful set of skills used to identify the underlying cause of issues is Systems Thinking.)

4. Select an approach to resolve the problem

- When selecting the best approach, consider:
- Which approach is the most likely to solve the problem for the long term?
- Which approach is the most realistic to accomplish for now? Do you have the resources? Are they affordable? Do you have enough time to implement the approach?
- What is the extent of risk associated with each alternative?

(The nature of this step, in particular, in the problem solving process is why problem solving and decision making are highly integrated.)



Get exactly 2 gallons of water into the 4gl jug.

Fig: 1.1

1. Entry Point

The *water_jugs* solution is derived by a simple, breadth-first, state-space search; and translated into a readable format by a DCG.

```
water_jugs :-
   SmallCapacity = 3,
   LargeCapacity = 4,
   Reservoir is SmallCapacity + LargeCapacity + 1,
   volume( small, Capacities, SmallCapacity ),
   volume( large, Capacities, LargeCapacity ),
   volume( reservoir, Capacities, Reservoir ),
   volume( small, Start, 0 ),
   volume( large, Start, 0 ),
   volume( reservoir, Start, Reservoir ),
   volume( large, End, 2 ),
   water_jugs_solution( Start, Capacities, End, Solution ),
   phrase( narrative(Solution, Capacities, End), Chars ),
   put_chars( Chars ).
```

water_jugs_solution(+Start, +Capacities, +End, ?Solution) holds when *Solution* is the terminal 'node' in a state-space search - beginning with a 'start state' in which the water-jugs have *Capacities* and contain the *Start* volumes. The terminal node is reached when the water-jugs contain the *End* volumes.

```
water_jugs_solution( Start, Capacities, End, Solution ) :-
   solve_jugs( [start(Start)], Capacities, [], End, Solution ).
```

solve_jugs(+Nodes, +Capacities, +Visited, +End, ?Solution) holds when *Solution* is the terminal 'node' in a state-space search, beginning with a first 'open' node in *Nodes*, and terminating when the water-jugs contain the *End* volumes. *Capacities* define the capacities of the water-jugs, while *Visited* is a list of expanded ('closed') node states.

The 'breadth-first' operation of solve_jugs is due to the 'existing' *Nodes* being appended to the 'new' nodes. (If the 'new' nodes were appended to the 'existing' nodes, the operation would be 'depth-first'.)

```
solve_jugs( [Node|Nodes], Capacities, Visited, End, Solution ) :-
   node_state( Node, State ),
   ( State = End ->
      Solution = Node
```

```
    ; otherwise ->
      findall(
        Successor,
        successor(Node, Capacities, Visited, Successor),
        Successors
        ),
      append( Nodes, Successors, NewNodes ),
      solve_jugs( NewNodes, Capacities, [State|Visited], End, Solution )
    ).
```

successor(+Node, +Capacities, +Visited, ?Successor) Successor is a successor of *Node*, for water-jugs with *Capacities*, if there is a legal 'transition' from *Node*'s state to *Successor*'s state, and *Successor*'s state is not a member of the *Visited* states.

```
successor( Node, Capacities, Visited, Successor ) :-
    node_state( Node, State ),
    Successor = node(Action,State1,Node),
    jug_transition( State, Capacities, Action, State1 ),
    \+ member( State1, Visited ).
```

jug_transition(+State, +Capacities, ?Action, ?SuccessorState) holds when *Action* describes a valid transition, from *State* to *SuccessorState*, for water-jugs with *Capacities*.

There are 2 sorts of *Action*:

- empty_into(Source,Target): valid if *Source* is not already empty and the combined contents from *Source* and *Target*, (in *State*), are not greater than the capacity of the *Target* jug. In *SuccessorState*: *Source* becomes empty, while the *Target* jug acquires the combined contents of *Source* and *Target* in *State*.
- fill_from(Source,Target): valid if *Source* is not already empty and the combined contents from *Source* and *Target*, (in *State*), are greater than the capacity of the *Target* jug. In *SuccessorState*: the *Target* jug becomes full, while *Source* retains the difference between the combined contents of *Source* and *Target*, in *State*, and the capacity of the *Target* jug.

In either case, the contents of the unused jug are unchanged.

```
jug_transition( State0, Capacities, empty_into(Source,Target), State1 ) :-
    volume( Source, State0, Content0 ),
    Content0 > 0,
    jug_permutation( Source, Target, Unused ),
    volume( Target, State0, Content1 ),
    volume( Target, Capacities, Capacity ),
    Content0 + Content1 =< Capacity,
    volume( Source, State1, 0 ),
    volume( Target, State1, Content2 ),
    Content2 is Content0 + Content1,
    volume( Unused, State0, Unchanged ),
    volume( Unused, State1, Unchanged ).
jug_transition( State0, Capacities, fill_from(Source,Target), State1 ) :-
    volume( Source, State0, Content0 ),
    Content0 > 0,
    jug_permutation( Source, Target, Unused ),
    volume( Target, State0, Content1 ),
    volume( Target, Capacities, Capacity ),
    Content1 < Capacity,
    Content0 + Content1 > Capacity,
    volume( Source, State1, Content2 ),
    volume( Target, State1, Capacity ),
```

```
    Content2 is Content0 + Content1 - Capacity,
    volume( Unused, State0, Unchanged ),
    volume( Unused, State1, Unchanged ).
```

2. Data Abstraction

volume(?Jug, ?State, ?Volume) holds when *Jug* ('large', 'small' or 'reservoir') has *Volume* in *State*.

```
volume( small, jugs(Small, _Large, _Reservoir), Small ).
volume( large, jugs(_Small, Large, _Reservoir), Large ).
volume( reservoir, jugs(_Small, _Large, Reservoir), Reservoir ).
```

jug_permutation(?Source, ?Target, ?Unused) holds when *Source*, *Target* and *Unused* are a permutation of 'small', 'large' and 'reservoir'.

```
jug_permutation( Source, Target, Unused ) :-
    select( Source, [small, large, reservoir], Residue ),
    select( Target, Residue, [Unused] ).
```

node_state(?Node, ?State) holds when the contents of the water-jugs at *Node* are described by *State*.

```
node_state( start(State), State ).
node_state( node(_Transition, State, _Predecessor), State ).
```

3. Definite Clause Grammar

narrative/5 is a DCG presenting water-jugs solutions in a readable format. The grammar is 'head-recursive', because the 'nodes list', describing the solution, has the last node outermost.

```
narrative( start(Start), Capacities, End ) -->
    "Given three jugs with capacities of:", newline,
    literal_volumes( Capacities ),
    "To obtain the result:", newline,
    literal_volumes( End ),
    "Starting with:", newline,
    literal_volumes( Start ),
    "Do the following:", newline.
narrative( node(Transition, Result, Predecessor), Capacities, End ) -->
    narrative( Predecessor, Capacities, End ),
    literal_action( Transition, Result ).

literal_volumes( Volumes ) -->
    indent, literal( Volumes ), ";", newline.

literal_action( Transition, Result ) -->
    indent, "- ", literal( Transition ), " giving:", newline,
    indent, indent, literal( Result ), newline.

literal( empty_into(From,To) ) -->
    "Empty the ", literal( From ), " into the ",
    literal( To ).
literal( fill_from(From,To) ) -->
    "Fill the ", literal( To ), " from the ",
    literal( From ).
literal( jugs(Small,Large,Reservoir) ) -->
    literal_number( Small ), " gallons in the small jug, ",
    literal_number( Large ), " gallons in the large jug and ",
    literal_number( Reservoir ), " gallons in the reservoir".
literal( small ) --> "small jug".
literal( large ) --> "large jug".
literal( reservoir ) --> "reservoir".
```

```
literal_number( Number, Plus, Minus ) :-
    number( Number ),
    number_chars( Number, Chars ),
    append( Chars, Minus, Plus ).

indent --> "  ".

newline --> "
".
```

4. Utility Predicates

Load a small library of puzzle utilities.

```
:- ensure_loaded( misc ).
```

5. Result (Output in Prolog)

The output of the program is:

```
?- water_jugs.

Given three jugs with capacities of:
  3 gallons in the small jug, 4 gallons in the large jug and 8 gallons in the reservoir; To obtain the result:  0 gallons in the small jug, 2 gallons in the large jug and 6 gallons in the reservoir; Starting with:  0 gallons in the small jug, 0 gallons in the large jug and 8 gallons in the reservoir;Do the following:
  - Fill the small jug from the reservoir giving:
    3 gallons in the small jug, 0 gallons in the large jug and 5 gallons in the reservoir
  - Empty the small jug into the large jug giving:
    0 gallons in the small jug, 3 gallons in the large jug and 5 gallons in the reservoir
  - Fill the small jug from the reservoir giving:
    3 gallons in the small jug, 3 gallons in the large jug and 2 gallons in the reservoir
  - Fill the large jug from the small jug giving:
    2 gallons in the small jug, 4 gallons in the large jug and 2 gallons in the reservoir
  - Empty the large jug into the reservoir giving:
    2 gallons in the small jug, 0 gallons in the large jug and 6 gallons in the reservoir
  - Empty the small jug into the large jug giving:
    0 gallons in the small jug, 2 gallons in the large jug and 6 gallons in the reservoir

Yes
```

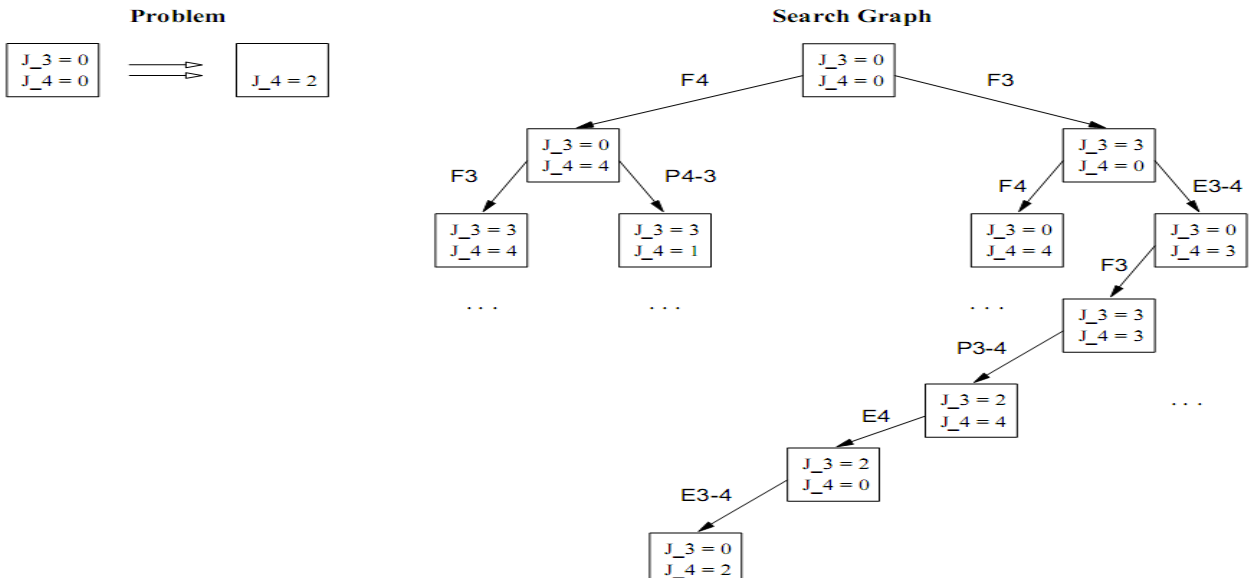


Fig:1.2

6. Conclusion

Problem solving is a mental process and is part of the larger problem process that includes problem finding and problem shaping. Considered the most complex of all intellectual functions, problem solving has been defined as higher-order cognitive process that requires the modulation and control of more routine or fundamental skills. Problem solving occurs when an organism or an artificial intelligence system needs to move from a given state to a desired goal state. Thus conclude to get 2 gallon water in 4 liter jug.

7. Reference

1. E. Rich & K. Knight, Artificial Intelligence, 2nd edition, McGraw-Hill, 1991
2. Groner, M., Groner, R., & Bischof, W. F. (1983). Approaches to heuristics: A historical review. In R. Groner, M. Groner, & W. F. Bischof (Eds.), Methods of heuristics (pp. 1-18). Hillsdale, NJ: Lawrence Erlbaum Associates.
3. Halpern, Diane F. (2002).Thought & Knowledge. Lawrence Erlbaum Associates. Worldcat Library Catalog Kluwe, R. H. (1993). Knowledge and performance in complex problem solving. In G. Strube & K.-F. Wender (Eds.), The cognitive psychology of knowledge (pp. 401-423). Amsterdam: Elsevier Science Publishers

4. Mayer, R. E. (1992). Thinking, problem solving, cognition. Second edition. New York: W. H. Freeman and Company.
5. Müller, H. (1993). Komplexes Problemlösen: Reliabilität und Wissen [Complex problem solving: Reliability and knowledge]. Bonn, Germany: Holos

6. Sternberg, R. J., & Frensch, P. A. (Eds.). (1991). Complex problem solving: Principles and mechanisms. Hillsdale, NJ: Lawrence Erlbaum Associates. Strauß, B. (1993). Konfundierungen beim Komplexen Problemlösen. Zum Einfluß des Anteils der richtigen Lösungen (ArL) auf das

Problemlöseverhalten in komplexen Situationen [Confoundations in complex problem solving. On the influence of the degree of correct solutions on problem solving in complex situations]. Bonn, Germany: Holos