A note on the water jug problem in Prodigy

Doug Beeferman

December 12, 1997

An instance of the *water jug problem* is a set of n distinct jugs $\{J_i\}$ with capacities C_i, an initial configuration, and a goal configuration. An initial configuration is an assignment v_i of jugs to starting volumes, and a goal configuration is a partial assignment g_i of jugs to goal volumes (with possible "don't cares".) We restrict our attention to problem instances in which the C_i, v_i, and g_i are all nonnegative integers. We shall abbreviate such a problem instance as $(v_1/C_1, v_2/C_2, \ldots, v_n/C_n) \rightarrow (g_1, g_2, \ldots, g_n)$. A classic example of the problem, cited in Rich and Knight [1], is the two-jug problem $(0/3, 0/4) \rightarrow (?, 2)$.

The operators in the water jug domain are fill(x), dump(x), and transfer(x, y), where all the arguments represent jug identifiers. The *fill* operator fills a jug to capacity with water from a faucet. The *dump* operator dumps all of the water from a jug. The *transfer* operator transfers as much of the water from x to y as it can, without y overflowing.

Two transfer operators

A transfer from jug i to jug j falls broadly into one of two classes: It can be a *complete* transfer of all the water in jug i to jug j, or it can be a *partial* transfer of some of the water in jug i to jug j. In the former case, jug i ends up empty and jug j ends up nonempty; in the latter case, jug i ends up nonempty and jug j ends up full.

An early implementation of the jug problem in the Prodigy problem-solving environment encoded two separate transfer operators, one to handle each of these cases. An important fact is that this split of the abstract transfer operator is unnecessary; that is, for a given jug configuration and an intent to transfer water from a fixed jug i to another jug j, it is always well-defined which of the two subcases, if any, will be applied: if $w_i = 0$ or $w_j = C_j$, then no transfer is possible; otherwise, if $w_i \leq C_j - w_j$, where w_i

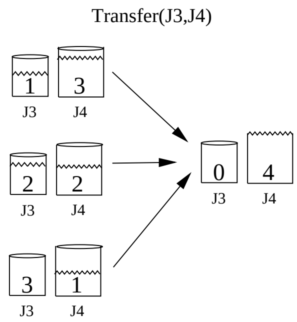

Figure 1: Jug configurations map many-to-one to each other under a fixed transfer operator.

is the level of water in jug i, then the first subcase applies; otherwise, the second subcase applies.

A fairly apparent fact is that the sum of the water levels in the jugs holds steady across transfer operator applications. Furthermore, it can be easily shown that if the initial state of the problem includes at most one non-empty, non-full jug, then at every state in any path of operator applications, there is always at leat one non-empty, non-full jug. We prove this by a simple inductive argument, with the initial state as the base case. Consider an operator application applied to a state in which the inductive hypothesis holds. A fill operator either reduces or holds steady the number of non-empty, non-full jugs. The same is true of the dump operator. A transfer(x, y) operator always produces either an empty jug (in the first subcase) or a full jug (in the second subcase.)

A final comment on the transfer operator is that, although there is a well-defined final configuration given an initial configuration, the converse does not hold. In other words, the mapping between configurations induced by a transfer operator is many-to-one (see Figure 1).

A unified transfer operator

In this section we suggest a rewritten transfer operator that employs all these facts. It also fixes a bug in the earlier implementation that caused the domain to be incomplete over the problem space, and does away with a number of inefficiencies, including the unnecessary "space-available-p" predicate that signals when a jug has room. The new operator is shown below in the Prodigy Description Language.

```
(operator TRANSFER
 (params <from-j> <to-j>)
 (preconds
  ((<from-j> JUG)
   (<to-j> (and JUG (diff <from-j> <to-j>)))

   (<total-from-w>
    (and WATER (gen-from-pred (total <from-j> <total-from-w>))))
   (<total-to-w>
    (and WATER (gen-from-pred (total <to-j> <total-to-w>))))

   (<from-w> (and WATER (gen-lte <total-from-w>)))
   (<to-w> (and WATER (gen-to-w <total-to-w> <from-w> <total-from-w>))))
  (and
   (is-water <to-j> <to-w>)
   (is-water <from-j> <from-w>)))
 (effects
  ((<from-w-new> (and WATER (srcamounteq <from-w-new>
 <from-w> <to-w>
 <total-to-w>)))
   (<to-w-new> (and WATER (destamounteq <to-w-new>
<from-w> <to-w>
<total-to-w>))))
  ((del (is-water <from-j> <from-w>))
   (del (is-water <to-j> <to-w>))
   (add (is-water <from-j> <from-w-new>))
   (add (is-water <to-j> <to-w-new>))))) ;;;primary effect
```

The preconditions permit all possible values for the initial level of water in the source jug, and then tailor the chosen value for the initial level of water in the destination jug to depend on the selected source value, as follows:

```
;; Integers i such that  1<=i<=x
(defun gen-lte (x)
  (if (<= x 0)
      nil
    (if (= x 1) (list x) (cons x (gen-lte (- x 1))))))

;; Generates possible values for the initial destination water level
(defun gen-to-w (total-to-w from-w total-from-w)
  (if (or (= from-w 0) (= from-w total-from-w))
      (cons 0 (gen-lte (- total-to-w 1)))
    (list 0)))
```

The effects form of the operator generates values for the levels of water in the source and destination after the transfer is complete. These, as described above, are uniquely determined by the configuration before the transfer, as follows:

```
;;; compute dest final amount
(defun destamount (src dest maxdest)
  (if (<= (+ src dest) maxdest) (list (+ src dest)) (list maxdest)))

;; compute src final amount
(defun srcamount (src dest maxdest)
  (if (<= (+ src dest) maxdest) (list 0) (list (- (+ src dest) maxdest))))

(defun srcamounteq (x src dest maxdest)
  (if (and (not (p4::is-variable-p src))
   (not (p4::is-variable-p dest))
   (not (p4::is-variable-p maxdest)))
      (if (not (p4::is-variable-p x))
  (if (= x (car (srcamount src dest maxdest)))
      (list x))
(srcamount src dest maxdest))))

(defun destamounteq (x src dest maxdest)
  (if (and (not (p4::is-variable-p src))
   (not (p4::is-variable-p dest))
   (not (p4::is-variable-p maxdest)))
      (if (not (p4::is-variable-p x))
  (if (= x (car (destamount src dest maxdest)))
      (list x))
(destamount src dest maxdest))))
```

Results

Table 1 shows the timing and plan quality results for various problems on the old and modified domains. Figure 2 shows the plan produced by the new domain for a typical problem. Unifying the transfer operator proved to enhance efficiency in almost all cases, and sometimes plan quality. Because the new domain eliminates the unnecessary generation of bindings and obviates the need for a special space-availability predicate, performance is much higher in general.

Problem	Old plan size	Old time	New plan size	New time
p1	6	1.96/34	**6**	**1.74/35**
p2	2	17.23/289	**2**	**1.23/14**
p3	**fail**	∞	**4**	**1.95/24**
p4	2	0.20/13	**2**	**0.40/14**
p5	3	1.60/35	**4**	**3.95/24**
p6	6	2.16/34	**3**	**2.22/21**
p7	6	0.72/33	**4**	**105.94/1771**
p8	5	487.63/11567	**6**	**6.9/99**

Table 1: Prodigy performance on the *old* jug domain (with two transfer operators) and the *new* jug domain (with one transfer operator), on the suite of canonical examples. The timings are specified as "(clock time/#nodes)", where clock time is in seconds. The problems were run on a DEC Alpha 3000.

Random problem generation

A program was produced to output random instances of the water jug problem conforming to the basic constraint that a jug's water level may not exceed its capacity. As parameters, the program takes the maximum permitted number of jugs and maximum permitted water level. Results of the old and new domains on these problems are summarized in Table 2.

Discussion

We have presented a brief overview of some enhancements made to jug domain implemented in Prodigy. The changes improve performance and plan quality in most cases, but more critically are complete with respect to the canonical test examples as well as a suite of randomly generated examples. (We do not provide a proof of general completeness in this note; proving which problem instances have solutions at all is a difficult undertaking.)

Writing this domain proved difficult because of expressive weaknesses in the Prodigy Description Language in treating infinite types such as integer water levels. The order in which bindings are made and the times at which they are applied—facts that are critical to correctness and performance—are not transparent from the syntax. While the expressive power of the language is undeniably vast (its allowance for infinite types at all is pioneering!) it remains more compelling to hand-code solutions to such discrete math puz-

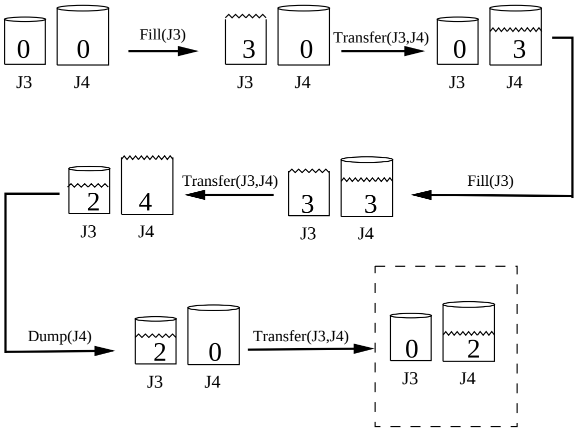

Figure 2: The plan produced by Prodigy for the problem $(0/3, 0/4) \rightarrow (?, 2)$

zles as the water jug problem as generic search algorithms in a more generic language, such as C, than to cast it as a planning problem.

References

[1] E. Rich and K. Knight *Artificial Intelligence: Section Edition* McGraw-Hill, Inc. 1991

Problem	**Old plan size**	**Old time**	**New plan size**	**New time**
$(0/1, 4/5) \rightarrow (?, 1)$	4	1.73/23	**5**	**2.10/30**
$(2/3, 4/4) \rightarrow (0, 3)$	4	2.74/69	**4**	**4.5/64**
$(1/4, 2/3, 2/2) \rightarrow (0, 2, 1)$	6	45.24/1168	**6**	**4.18/36**
$(5/5, 1/2) \rightarrow (?, 2)$	2	0.06/12	**1**	**0.03/8**
$(4/4, 2/5) \rightarrow (4, ?)$	0	0.00/5	**0**	**0.00/5**
$(2/4, 4/5) \rightarrow (?, 2)$	2	56.95/1071	**2**	**1.69/14**
$(2/2, 1/3, 1/5) \rightarrow (2, 0, 1)$	6	8.48/270	**6**	**133.29/2207**
$(5/6, 1/3) \rightarrow (1, 5)$	0	0.0/5	**0**	**0.00/5**

Table 2: Prodigy performance on 8 randomly generated problems for the *old* jug domain (with two transfer operators) and the *new* jug domain (with one transfer operator).