Measuring With Jugs: A Solution in Finite Domain

Charles R Ritchey, Jr., Russ Abbott, Eun-Young Kang
ritcheyc@gmail.com, {rabbott, eykang}@calstatela.edu
California State University, Los Angeles
Department of Computer Science
5151 State University Dr.
Los Angeles, CA 90032 USA
Tel (323)445-2792 Fax (323)343-6672

Abstract

The "measuring with jugs problem" involves pouring water between jugs to measure out a specific quantity of water. The object is to find the proper sequence of fill operations, pour operations (from one jug to another) and emptying operations to arrive at the desired quantity of water which may be in either jug. In the past, solutions to this puzzle have always involved creating a series of operations by computing various possible paths and choosing the one that leads to a solution. This paper takes a different, novel route to the solution to this. An answer is found numerically using constraint programming and the results are then translated into a path. At first, a simplified version of this puzzle using only two jugs will be discussed. A mathematical solution to this problem will be presented and the solution will then be translated into code in the Mozart-Oz programming language.

Keywords: Constraint Programming, Mozart-Oz, Finite Domain

1. Introduction

Mozart-Oz, more properly known as Oz3, is a multi-paradigm programming language. It may be used to create programs of many types and styles, i.e. many paradigms. Its main strength, though, is its ability to make short work of some rather complex programming problems using constraint programming. In standard logic programs, the computer must try multiple values for each variable until all values satisfy a set of conditions and a solution is reached. With constraint programming, however, the computer will solve a problem by taking a range of possible values for variables and narrowing their possible ranges based on constraints given by the programmer. The ranges of possible values are continually narrowed by the computer until a solution is found. This makes it excellent for finding the number of and type of operations necessary to solve this particular problem.

1.1 The Problem

In this simple version, the challenge is that given 2 jugs, one must measure out an exact quantity of water. This may be any quantity which is less than the volume of the largest jug. In this problem, there is an unlimited supply of water available for filling either jug or both jugs. Also, when a jug is filled, it must be filled to capacity. Either or both may be emptied as often as desired and in any order desired. The object is to find the proper sequence of fill operations, pour operations (from one jug to another) and emptying operations to arrive at the desired quantity of water which may be in either jug.

An example of this would be: given a 5 liter jug and a 3 liter jug, measure out 4 liters. A solution would be: Fill the 5 liter jug, pour 3 liters into the smaller jug filling it to capacity. Empty the smaller jug and pour the 2 liters remaining in the 5 liter jug into the 3 liter jug.

Fill the 5 liter jug. Pour 1 liter from the 5 liter jug into the 3 liter jug (which already contains 2 liters) to fill it to capacity. There are now 4 liters of water in the large jug (and 3 liters in the smaller jug).

```
%%
%% Jugs
%%
declare
proc {Jugs Root}
   CapA CapB TimesA TimesB TimesANeg TimesBNeg
in
   CapA=3
   CapB=5

   Root = sol(timesA:TimesA timesANeg:TimesANeg timesBNeg:TimesBNeg timesB:TimesB)
   Root ::: 0#9
   %% Negative values (jugs that get emptied into)are in TimesANeg and TimesBNeg
   %% value behind propigator is the final volume you want after all operations
    CapA*TimesA-CapA*TimesANeg+CapB*TimesB-CapB*TimesBNeg=:4
   {FD.distribute ff Root}
end

{Browse {SearchOne Jugs}}
```

Figure 1 -- Jugs

1.2 Analysis

This problem can be broken down to simple algebra. Since by definition, water may be poured between the jugs without restriction, there is no difference whether a volume of water is in which jug. The only real limiting factor is the final volume of water in the jugs. So the problem may simply be stated as

Capacity1*XNumber fill operations – Capacity2*YNumber emptying operations = Final Volume
Or
Capacity2*Y1Number fill operations – Capacity1*X1Number emptying operations = Final Volume

This can be translated into simple English as fill jug with capacity1 X number of times and pour into jug with capacity2 Y number of times emptying whenever filled to capacity *or* Fill Cap2 Y1 Number of times and empty into Cap1 X1 number of times, again emptying the jug when full. The order in which pours are made does not matter since the only important quantity is final volume.

While this paper was being researched, it was discovered that this idea had already been explored and that the jugs problem has been a common one both in computer science and in mathematics. One example was the “Measuring With Jugs”[1] in which the authors sought not only to solve the problem but also to analyze its complexity.

Perhaps, though, the clearest explanation was contained in the paper, “N Jugs and Water Problem”[4] The authors laid down a clear and concise mathematical argument similar to the one given above. This uncomplicated analysis of the jugs problem is not new.

1.3 Two Jugs

Since the problem may be reduced to such simple algebra, it follows that the solution should also be simple. Figure 1 contains the first edition of this program, Jugs.oz. It is based on the Send More Money program from the Mozart - Oz tutorial. [2]

Variables are used for both number of fills (TimesA) and empties (TimesANeg) for each jug. This is to allow the program to decide which path will be more efficient. The ff option (first fail) is used here to tell the computer to use the smallest values possible. This will guarantee that the smallest possible non negative integer for the values of TimesA, TimesANeg, TimesB, and TimesBNeg will be found. This should guarantee that a non zero number will not be found for both TimesX, number of fills, and TimesXNeg, number of empties.

As in the “Send More Money” solution[3], a record, Root, is used to hold variables. This enables all variables to be set simultaneously as “first fail”, and to limit their values to 0 and 9 and the integers between these values.

1.4 Many Jugs

In the 2 jugs version of this problem, the capacities of the jugs must be relatively prime. That is, they may not share a common factor. If the capacities of the jugs do share a common factor, the contents may not be measured to less than that factor. For example, if one of the jugs has a capacity of 9 while the smaller has a capacity of 6, the two share a common factor, 3. A person could not measure out less than 3 liters by any combination of pours from these jugs.

A new problem and its solution was suggested by two students. If you have more than 2 jugs, and none of them are relatively prime, 6 10 15, for example, a solution could still be found by "partitioning" the jugs into groups. The idea is this. No matter which jug you pick, the capacity has factors with the capacities of both of the others. By grouping jugs together as one, a "virtual" jug, one could get around this problem. For example, if jugs 10 and 6 were treated as one jug, the resulting capacity is 16. Since this and the other capacity, 15, are relatively prime, a solution can be found. Of course, the above restriction still holds if *all* of the jugs share a common factor.

1.5 Analysis

Again, this program can be broken down algebraically. Remember that addition is commutative, the order of addition does not matter. The order of subtraction also does not matter if you treat A - B as A+(-B). No matter the order, the result will be the same. Thus the order of the fills and empties does not matter provided that you don't try to pour from an empty container and jugs are not over filled.

The multiple jugs problem may again be simplified to

```
Cap A*(±NumPours A)+Cap B*(±NumPours B) ... =Volume
```

Where Cap is the capacity of a jug and NumPours is the number of operations on that jug. A positive NumPours indicates fill operations while a negative NumPours indicates number of times to empty into this jug. A solution could be found simply by filling jugs with positive NumPours and pouring into those with negative NumPours. The solution may be complicated by the fact that more than one jug may have contents at the end of these operations. A simple solution to this is simply to combine contents of all jugs into the largest after all fill, pour, and empty operations have occurred.

2. Finite Domain

The challenge here was to not only find numbers for a solution but also to produce an actual sequence of operations. As before, constraint programming, was used to calculate the number of fills, pours and empties. Now, however, a new set of functions had to be written to translate the set of numbers generated into a series of operations. Care was taken to ensure that all functions were as expandable as possible.

Figure 2 is the last version of the finite domain portion of this program. As before, the capacities of the jugs are stored in the CapX variables. Again, it uses a record, Sol. In this version, however, the variables are stored in an array, Vars. This array is then populated with finite domain integers using an FD.decl statement in a ForAll loop. In this version, 2 new variables have been introduced, Pours and Goal. Goal is simply the final volume after all operations. Pours is the number of pouring operations. This variable is minimized to limit the number of pours. Sol, the record is used primarily for output. This enables a user to see the actual values calculated by the program before a path was created.

```
159  fun {Jugs CapA CapB CapC Goal}
160     TimesA TimesB TimesC TimesANeg TimesBNeg TimesCNeg
161     %% List Vars holds the values of the numbers of pours
162     Vars = [TimesA TimesB TimesC TimesANeg TimesBNeg TimesCNeg]
163     Pours
164     %% Record Sol also holds variables -- this will be convenient for output
165     Sol = sol(goal:Goal pours:Pours
166  .            aCap:CapA aTimesA:TimesA aTimesANeg:TimesANeg
167  .            bCap:CapB bTimesBNeg:TimesBNeg bTimesB:TimesB
168  .            cCapC:CapC cTimesC:TimesC cTimesCNeg:TimesCNeg)
169     XTimesA XTimesB XTimesC Path
170  in
171     {ForAll Pours|Vars proc {$ Var} {FD.decl Var} end}
172
173     {FD.distribute ff [Pours]}
174     {FD.sum Vars '=:' Pours}
175
176     CapA*(TimesA-TimesANeg) + CapB*(TimesB-TimesBNeg) + CapC*(TimesC-TimesCNeg) =: Goal
177
178     {FD.distribute ff Vars}
179     %% Xtimes is either positive or negative
180     XTimesA=TimesA-TimesANeg
181     XTimesB=TimesB-TimesBNeg
182     XTimesC=TimesC-TimesCNeg
183
184     {WritePath CapA#XTimesA#0 CapB#XTimesB#0 CapC#XTimesC#0 Path}
185     Sol#Path
186  end
```

Figure 2 -- function Jugs

This function may be expanded for other jugs with a few changes.

1 For each jug, add an extra CapX variable to the input (line 2).
2 Add TimesX and TimesXNeg to variables (3)
3 Alter the record. For each jug, add xCapX:CapX xTimesX:TimesX and xTimesXNeg:TimesXNeg(4)
4 AddTimesX and TimesXNeg toe Vars (4)
5 Add "X"TimesX to the list of variables
6 Add a CapX+(TimesX-TimesXNeg) to the following line:
CapA*(TimesA-TimesANeg) + CapB*(TimesB-TimesBNeg) + CapC*(TimesC-TimesCNeg) =: Goal
7 Add "X"TimesX=TimesX-TimesXNeg
8 Lastly add 'X'TimesX to function call to WritePath . (This is not shown in this listing. It would be a close to the end of this function.)

Listed below are the functions that help to write a path for from the finite domain data. The last one listed, WritePath, is the main program. The rest are support functions that could have been made sub functions.

MaxCap -- finds jug with greatest capacity
Remove -- Removes a jug from a list and returns the remaining list
GCap -- uses MaxCap and Remove to find jug with maximum capacity and returns the rest
Greatest -- finds jug with greatest contents
ZeroList – checks the XTimes values for list members. If all are Zero, returns TRUE else returns FALSE
SumList3 -- Returns sum of contents of list of jugs
Combine2 -- writes a path that pours all jugs into largest
Combine -- Uses GCAP, Sumlist3, and Combine2 to combine all jugs into largest and write a path
SmallestXTimes -- finds smallest XTimes
SmallestPos -- returns smallest positive XTimes
WritePath -- Uses previous functions to Write a path describing all operations
Jugs -- finite domain portion of program to find numbers

The operation of the functions of this program, while apparently complex is actually quite simple. First, Write Path would

be called to generate a path. ZeroList would check to make sure that the jugs had not become depleted. If so, any remaining would be combined into one and the program would exit. If not, Smallest would find the jug to be emptied into. SmallPos would find the jug that would be pouring. The function DoSomething, is where the actual operations were performed. Here, also, is where the operations were recorded and appended to the variable Path.

The complete programs can be found at http://cs.calstatela.edu/%7Ewiki/images/e/e7/JugsExten.oz and http://cs.calstatela.edu/%7Ewiki/images/8/84/Jugs7.oz.

3. Conclusion

This program demonstrates that constraint programming may be used for some applications where is might not seem useful at first glance. Finding a series of operations in the jugs problem, while not apparently simple, may be broken down mathematically and then easily solved.

References

[1] P. Boldi, M. Santini, S. Vigna, "Measuring With Jugs", Theoretical Computer Science 282(2), 2002. (Viewable online at http://vigna.dsi.unimi.it/ftp/papers/Jugs.pdf)

[2] C. Schulte, G. Smolka, "Finite Domain Constraint Programming in Oz: a Tutorial". (Viewable online at http://www.mozart-oz.org/documentation/fdt/node15.html#section.problem.moneyy)

[3] M Tran, T Pfaf "The N-Jugs and Water Problem", 2005