A COMPUTER PROGRAM TO SOLVE WATER JUG POURING PUZZLES

RICHARD J. MATHAR

Abstract. We provide a C++ program which searches for the smallest number of pouring steps that convert a set of jugs with fixed (integer) capacities and some initial known (integer) water contents into another state with some other prescribed water contents. Each step requires to pour one jug into another without spilling until either the source jug is empty or the drain jug is full—because the model assumes the jugs have irregular shape and no further marks.

The program simply places the initial jug configuration at the root of the tree of state diagrams and deploys the branches (avoiding loops) recursively by generating all possible states from known states in one pouring step.

1. Liquid Pouring: The Rules

1.1. **Standard Puzzle.** The grand father of the liquid pouring puzzles comes like this: Supposed there are three water jugs with capacities of 8, 5 and 3—in some units like liters or gallons. The largest is filled with water to capacity and the other two are empty. The jugs have irregular shapes and no further volume marks. How does one get 4 units into one of them by pouring water from one jug to another? How many of these pouring steps are needed at minimum?

The rules of pouring water from any jug into another derive from the fact that there are two implicit marks on each jug, the mark of its capacity (full) and the mark of 0 (empty). Pouring exchanges liquid until either the source jug gets empty or until the target jug is full, whichever happens first. If we note the liquid content of the 3 jugs of capacities (8,5,3) in that order as a triple of integers, one solution of the problem is [8,0,0] → [3,5,0] → [3,2,3] → [6,2,0] → [6,0,2] → [1,5,2] → [1,4,3]. In plain words: This means 7 steps suffice to get 4 units into the jug of capacity 5: first fill the jug of capacity 5 to the rim; then pour from the jug with capacity 5 into the jug of capacity 3 to the rim; then pour all water from the jug with capacity 3 into the jug of capacity 8; then pour all water from the jug with capacity 5 into the jug of capacity 3; then fill the jug of capacity 5 to the rim from the jug of capacity 8 (again); finally pour from the jug of capacity 5 to the jug of capacity 3 up to the rim. This leaves a state where the middle size jug holds 4 units.

1.2. **Volume Exchanged.** The rule of pouring liquid from a "source" jug with capacity c_s ($1 \le s \le 3$, the jug number) containing f_s units ($0 \le f_s \le c_s$) to a "destination" jug with capacity c_d ($1 \le d \le 3$, the jug number) containing f_d units

Date: September 16, 2015.

2010 *Mathematics Subject Classification.* Primary 00A08; Secondary 97A80, 90C35.

Key words and phrases. Puzzles, Water Jugs, Pouring.

requires to halt when the jug s is empty or jug d is full. This transfers either f_s or $c_d - f_d$ units, which may be written as

$$t_{s\to d} = \min(f_s, c_d - f_d) \tag{1}$$

units. In the puzzle such a step makes only sense if that value is positive, $t > 0$, because a step that does not actually change the jugs' contents would be a waste of steps. In our bracket notation this is a step $[\dots, f_s,\dots f_d] \to [\dots, f_s - t,\dots f_d + t]$.

Remark 1. *Because the exchanged volume t is a unique function of the capacities and filling states, one could actually compress the notation of the steps by writing down only the two indices of the source and destination jug at each step to express the actions.*

The steps are not necessarily reversible, i.e, it may be impossible to undo the liquid exchange t by swapping the roles (indices s and d) under some circumstances [8]. Consider for example two jugs with capacities (5,3) initially filled with [1,2], then pouring all of the bigger jug—1 unit—into the smaller, so they are filled with [0,3] afterwards. The rules then forbid to pour the 1 unit from the smaller jug back into the larger jug, because neither the smaller jug nor the larger jug has a mark to stop after 1 unit. (The states do not have memory, so to speak.) The requirements for a reversible step can be summarized as follows by equating the two values t of the pouring and its reverse:

Theorem 1. *A pouring is reversible if (and only if) either the source jug is full or the destination jug is empty (or both).*

Remark 2. *Because the capacities c_j of the jugs do not change while pouring, one could also rephrase this as an exchange of air volumes*

$$a_j \equiv c_j - f_j \tag{2}$$

between pairs of jugs—similar to the introduction of hole states instead of electrons in semiconductor physics. This rephrasing does not change the nature of the puzzle.

In any case the total capacity

$$C \equiv \sum_{j=1}^{J} c_j \tag{3}$$

of the J jugs, the total water content

$$F \equiv \sum_{j=1}^{J} f_j \tag{4}$$

and the total air content

$$A \equiv \sum_{j=1}^{J} a_j \tag{5}$$

stay constant.

1.3. **2 Jugs Puzzles.** The discussion above exchanges water between jugs without spilling; the total content $\sum_i f_i$ of all jugs is constant through all the steps.

There is a variant of the puzzle which states the following problem: given two jugs of capacity 5 and 3, plus a water tap as an infinite source and a drain which swallows any amount of water, how can 4 units be put into the jug of capacity 5? We realize that a solution is found by striking the filling state of the jug of capacity 8 from the solution of Section 1.1: [0,0] → [5,0] → [2,3] → [2,0] → [0,2] → [5,2] → [4,3]. This means: fill the large jug from the tap; then fill the small jug from the large jug; pour the small jug into the drain; pour all of the large jug into the small jug; fill the large jug from the tap; finally fill the small jug from the large jug. We have actually replaced pouring water into our out of the jug of capacity 8 in Section 1.1 by pouring water into the drain or refilling from the tap.

Solving such a puzzle with 2 jugs and infinite source and drain is equivalent to a puzzle with 3 jugs as follows: Define the initial air content of the third jug to be the total liquid content of the other jugs, $a_3 = \sum_{j<3} f_j$. Define initial liquid content of the third jug to be the total air content of the other jugs, $f_3 = \sum_{j<3} a_j$. Necessarily the capacity of the third jug equals the sum of the capacities of the other jugs, $c_3 = a_3 + f_3 = \sum_{j<3} c_j$, and a total liquid volume in all three jugs is the same, $F = \sum_j f_j = f_3 + \sum_{j<3} f_j = \sum_{j<3} a_j + \sum_{j<3} f_j = \sum_{j<3} c_j$.

This ensures that the additional jug can play the roles both as the tap and as the drain in a sort of recycling buffer of sufficient capacity:

- Whatever was poured into the drain in the 2 jugs puzzle can be poured into the large jug in the 3 jugs puzzle, because its capacity is large enough to be refilled with the full content of any of the 2 smaller jugs. (Note that frequent refilling in the 2 jugs puzzle can be served in the 3 jugs puzzle from the large jug because everything that is poured into the drain in the 2 jugs puzzle returns to the large jug in the 3 jugs puzzle.)
- Whatever was coming out of the tap in the 2 jugs puzzle can be poured out of the large jug in the 3 jugs puzzle because its initial content was large enough to comply with a full filling of the smaller jugs.
- The next concern is whether the third jug adds transitions that have no counterpart in the 2 jugs puzzle. Is it possible for example that the third jug is at some time filled with 4 units which are poured into the jug of capacity 5, which is impossible in the 2 jugs puzzle, which might add artificial solutions to the problem? This scenario actually cannot occur.

  Proof. Supposed we would pour the 4 units from the jug with $c = 8$ to the jug with $c = 5$—which does not change the total liquid volume F. Afterwards the jug with $c = 8$ is empty and the destination jug with $c = 5$ is not full. The total liquid volume F in all jugs therefore is the liquid volume only in the two smaller jugs, $F = \sum_{j<3} f_j$. This volume is less than the total capacity of the two smaller jugs because at least one of the smaller jugs is not full, $F < \sum_{j<3} c_j$. And this contradicts the construction shown above which maintains $F = \sum_{j<3} c_j$ at all time. □

- The remaining concern is that the solution with a minimum number of steps that results with 4 units in any jug ends up with a state where the 4 units are in the largest jug. Apparently these spurious solutions of the 3 jug puzzle which have no counterpart in the 2 jug puzzle cannot be avoided;

so the puzzle must be formulated with the constraint that the 4 units of liquid end up in the jug of capacity 5 to have equivalent transitions in both puzzles.

For the rest of the paper we assume that only "closed" systems—with a fixed total amount F of liquid—will be modeled.

2. Available Literature

2.1. **Barycentric Coordinates.** Because the total liquid volume F is constant for all pouring steps, one can map the transitions to a type of billiard motions in barycentric coordinates [4, 12, 1][3, §4.6].

2.2. **Modular Divisions.** From a number-theoretic point of view, filling a smaller jug from the content of a larger jug is similar to computing a remainder of an integer division in the larger jug. In addition we may replace each jug by an infinitely large one with marks at each multiple of its original capacity. The generates an algebraic problem in congruences, akin to Euler's algorithm of computing greatest common divisors, and helps for example to put the water jugs puzzles in solvable and unsolvable categories [11] [10, §7.1][9].

3. State Diagrams

3.1. **Definition.** A visual overview of which volume shares can be produced is a state digram: each set of filling states of a given set of capacities and a given total amount of liquid F is a node in a labeled digraph (in the mathematical sense) [5][6, §1.2]. As vertex labels we shall use the tuple $[f_j]$ of the liquid content of the jugs sorted along decreasing c_j, $1 \le j \le J$ where J is the number of jugs. Pairs of states are connected by a directed edge if the state at the head of the edge can be reached by the state at the tail of the edge in one pouring step. As a visual aid we paint pairs of edges in green if the two states are reversible in the sense of Section 1.2.

The maximum outdegree of a vertex is $J(J-1)$ because this is the number of pairs of jugs that can exchange liquid.

The example of Figure 1 shows the graph for jugs of capacities 8, 5 and 3 and total liquid content $F = 4$. In this case the three vertices labeled with filling [1,1,2], [1,2,1] or [2,1,1] have indegree zero, which means they cannot be reached from any other partition of the 4 (unless they are the initial states).

Also one may move from [0,2,2] to [1,0,3] passing through [2,0,2] or [0,1,3], but the other direction cannot be done. Actually one can move around within the big cluster of states with 0+1+3 or 0+0+4 liquid units, and within the smaller cluster of 0+2+2 units, but one cannot reach any of the smaller cluster states from the big cluster (and one cannot reach the isolated states at all).

3.2. **Double-Edge circuits.** We observe that the reversible states always appear in rings (circuits), which means if a state can reach a second state reversibly in a single step, it can also reach at least one further state reversibly in a single step:

Theorem 2. *The subgraph which contains all the green double edges (collapsed into single undirected edges) does not contain isolated nodes with degree 1.*

Proof. A state reaches another state reversibly if either the source jug s is full or the destination jug d is empty: Theorem 1.

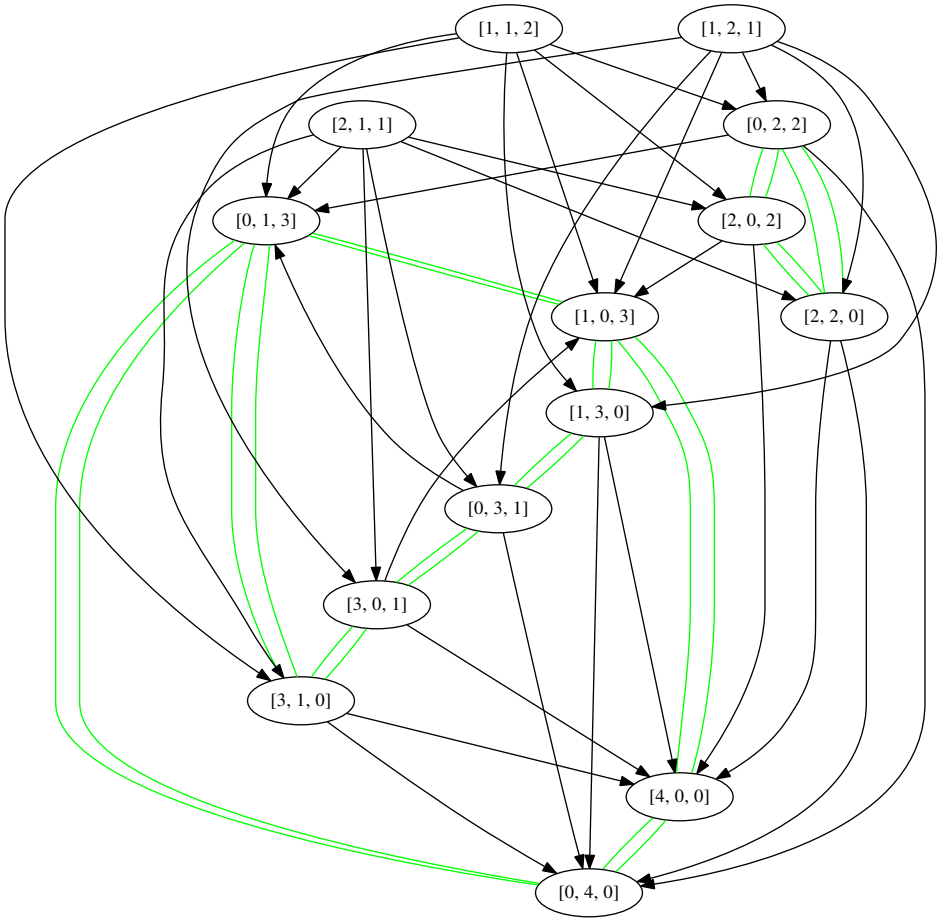

FIGURE 1. All states with 3 jugs of capacities (8,5,3) and a total liquid volume of 4. If pouring between two states is reversible two green edges connect the two states.

- If the destination jug is empty, there are $J-2$ alternative source jugs which are compatible with the requirement. So there is at least one alternative s' of a reversible step filling s' into d if there are $J \geq 3$ jugs. If this is impossible because these alternative source jugs are all empty, one may instead take any of them as an alternative destination jug and pour the source jug into that one.
- If the source jug is full it can be poured reversibly into any other of the $J-2$ destination jugs d' to fulfill the requirement. If this is impossible because these alternative destination jugs are all full, one may instead take any of them as an alternative source jug and pour it into the destination jug.

□

	[0,5,3]	[1,4,3]	[1,5,2]	[2,3,3]	[2,4,2]	[2,5,1]	[3,2,3]	[3,3,2]	[3,4,1]	[3,5,0]	[4,1,3]	[4,2,2]	[4,3,1]	[4,4,0]	[5,0,3]	[5,1,2]	[5,2,1]	[5,3,0]	[6,0,2]	[6,1,1]	[6,2,0]	[7,0,1]	[7,1,0]	[8,0,0]
[0,5,3]	0	6	5	3	-1	4	2	-1	-1	1	7	-1	-1	7	1	-1	-1	2	4	-1	3	5	6	2
[1,4,3]	1	0	1	3	-1	4	3	-1	-1	2	2	-1	-1	1	1	-1	-1	2	2	-1	3	4	3	2
[1,5,2]	1	1	0	4	-1	5	2	-1	-1	1	3	-1	-1	2	2	-1	-1	3	1	-1	2	5	4	2
[2,3,3]	1	6	6	0	-1	1	3	-1	-1	2	4	-1	-1	5	1	-1	-1	1	5	-1	4	2	3	2
[2,4,2]	2	1	1	1	0	1	3	-1	-1	2	2	-1	-1	1	2	-1	-1	2	1	-1	2	2	3	2
[2,5,1]	1	5	5	1	-1	0	2	-1	-1	1	3	-1	-1	4	2	-1	-1	2	4	-1	3	1	2	2
[3,2,3]	1	4	3	3	-1	4	0	-1	-1	1	6	-1	-1	5	1	-1	-1	2	2	-1	1	5	6	2
[3,3,2]	2	2	1	1	-1	2	1	0	-1	1	4	-1	-1	3	2	-1	-1	1	1	-1	2	3	4	2
[3,4,1]	2	1	2	2	-1	1	1	-1	0	1	2	-1	-1	1	2	-1	-1	3	3	-1	2	1	2	2
[3,5,0]	1	5	4	4	-1	5	1	-1	-1	0	7	-1	-1	6	2	-1	-1	3	3	-1	2	6	7	1
[4,1,3]	1	2	3	3	-1	3	3	-1	-1	2	0	-1	-1	1	1	-1	-1	2	4	-1	4	2	1	2
[4,2,2]	2	2	1	4	-1	4	1	-1	-1	2	1	0	-1	1	2	-1	-1	3	1	-1	1	3	2	2
[4,3,1]	2	2	3	1	-1	1	3	-1	-1	2	1	-1	0	1	2	-1	-1	1	4	-1	4	1	2	2
[4,4,0]	2	1	2	4	-1	4	2	-1	-1	1	1	-1	-1	0	2	-1	-1	3	3	-1	3	3	2	1
[5,0,3]	1	7	6	2	-1	3	3	-1	-1	2	6	-1	-1	7	0	-1	-1	1	5	-1	4	4	5	1
[5,1,2]	2	2	1	2	-1	3	3	-1	-1	2	1	-1	-1	2	1	0	-1	1	1	-1	2	2	1	2
[5,2,1]	2	4	3	2	-1	1	1	-1	-1	2	3	-1	-1	4	1	-1	0	1	2	-1	1	1	2	2
[5,3,0]	2	6	5	1	-1	2	2	-1	-1	1	5	-1	-1	6	1	-1	-1	0	4	-1	3	3	4	1
[6,0,2]	2	2	1	3	-1	4	2	-1	-1	2	4	-1	-1	3	1	-1	-1	2	0	-1	1	5	5	1
[6,1,1]	2	3	2	2	-1	1	2	-1	-1	2	1	-1	-1	2	2	-1	-1	3	1	0	1	1	1	2
[6,2,0]	2	3	2	4	-1	5	1	-1	-1	1	5	-1	-1	4	2	-1	-1	3	1	-1	0	6	6	1
[7,0,1]	2	4	5	2	-1	1	3	-1	-1	2	2	-1	-1	3	1	-1	-1	2	5	-1	4	0	1	1
[7,1,0]	2	3	4	3	-1	2	2	-1	-1	1	1	-1	-1	2	2	-1	-1	3	4	-1	3	1	0	1
[8,0,0]	2	6	5	3	-1	4	2	-1	-1	1	7	-1	-1	7	1	-1	-1	2	4	-1	3	5	6	0

TABLE 1. The distances between node pairs in Figure 2. The rows are labeled with the initial state; the columns are labeled with the final state. The distance is the minimum number of steps, -1 if the final state cannot be reached.

Remark 3. *Rem and Choo appear to conclude that (at least for the 3 jugs problem) solutions (albeit not the shortest ones) can always be found by moving only along the green edges* [7].

3.3. **Examples.** The standard puzzle of having $F = 8$ units and generating by any means 4 units is illustrated in Figure 2. One realizes that the [3,4,1] and [4,2,2] and [4,3,1] states are unreachable (indegree zero) but a path from [8,0,0] to [4,4,0] exists via [3,5,0], [3,2,3], [6,2,0], [6,0,2], [1,5,2] and [1,4,3] for example.

The number of steps of reaching a final state for any given initial state is summarized in Table 1.

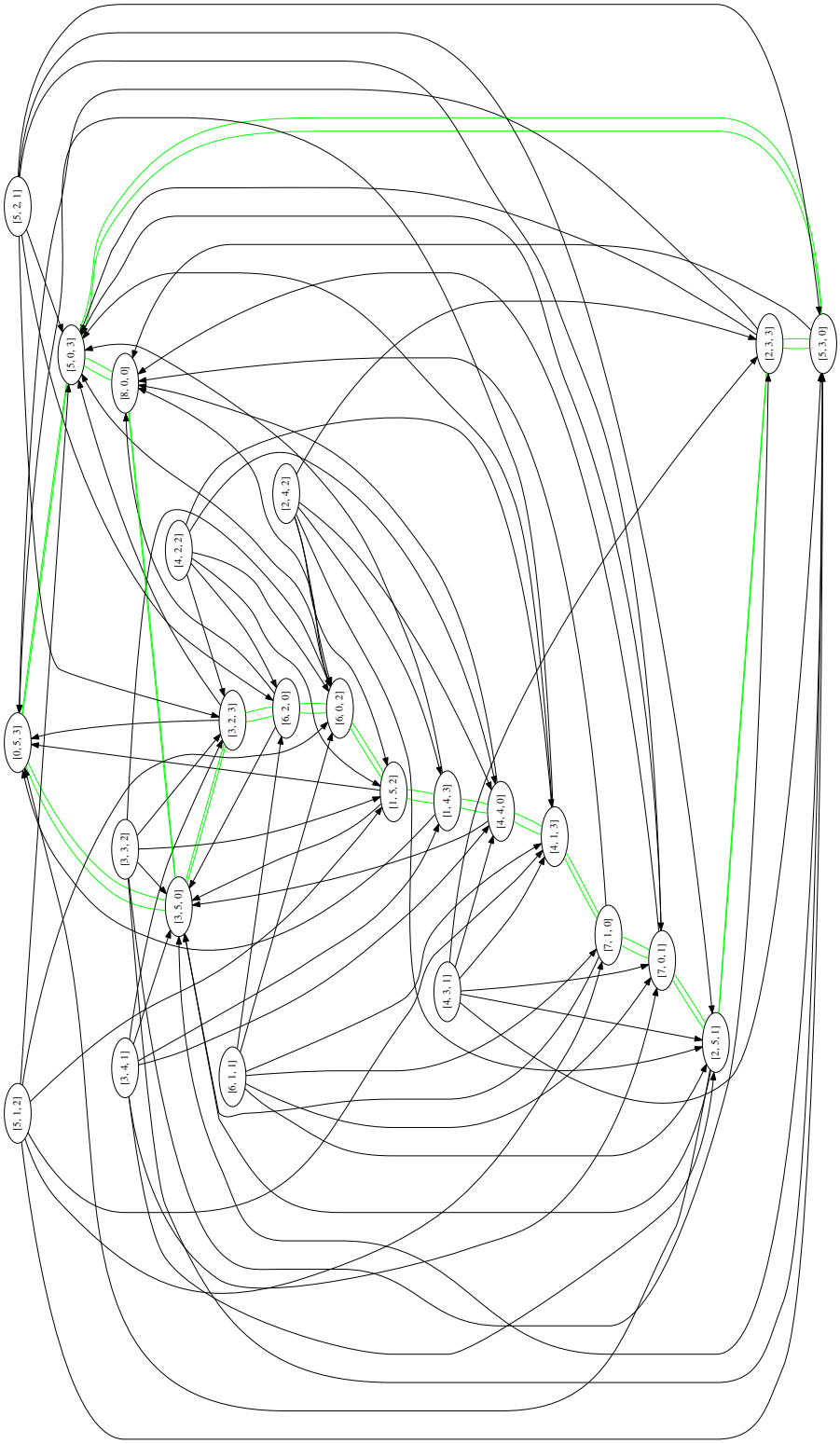

FIGURE 2. All 24 states with 3 jugs of capacities (8,5,3) and a total liquid volume of 8.

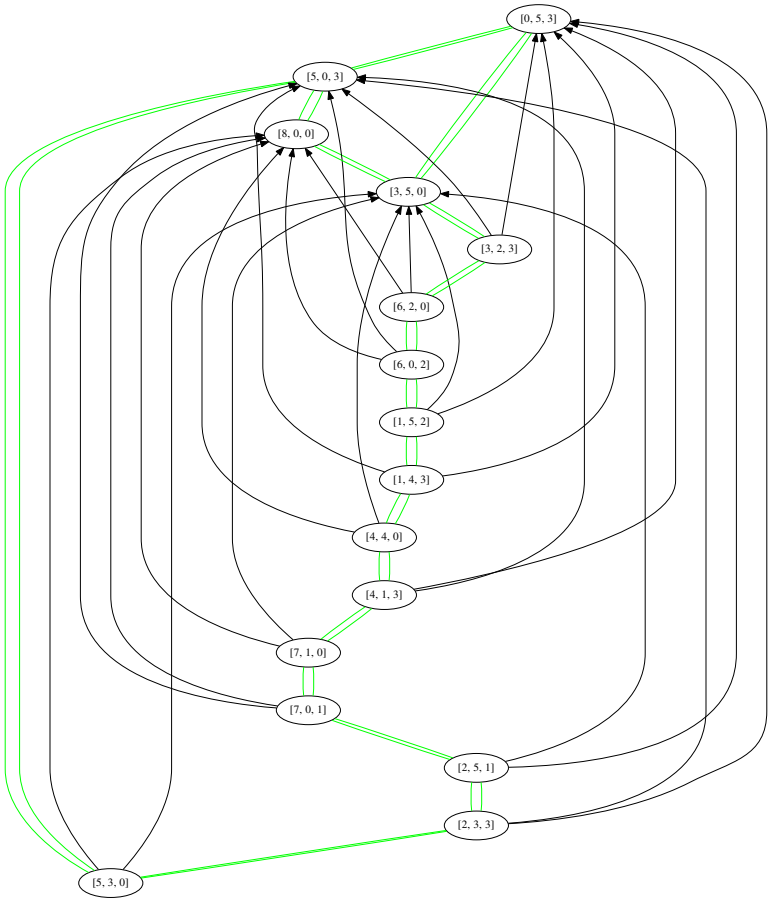

FIGURE 3. The 16 states with 3 jugs of capacities (8,5,3) and a total liquid volume of 8 obtained from Figure 2 by removing all vertices with zero indegree.

If we remove those unreachable states from Figure 2 we obtain figure 3.

If the largest jug is replaced by a jug of 6 units of capacity and also the total liquid volume reduced to 6 units, the state diagram of Figure 4 arises.

If the largest jug is replaced by a larger jug of 12 units of capacity and also the total liquid volume is 12 units, the state diagram of Figure 5 arises.

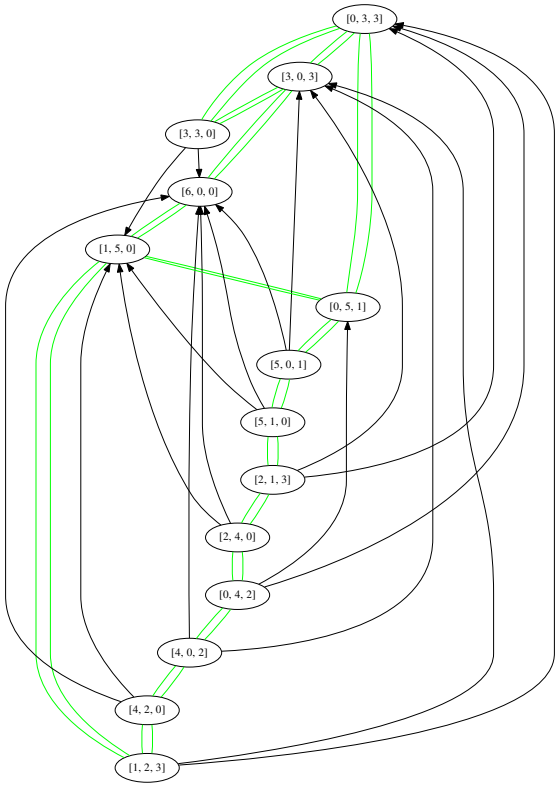

FIGURE 4. The states with 3 jugs of capacities (6,5,3) and a total liquid volume of 6 after removing all vertices with zero indegree.

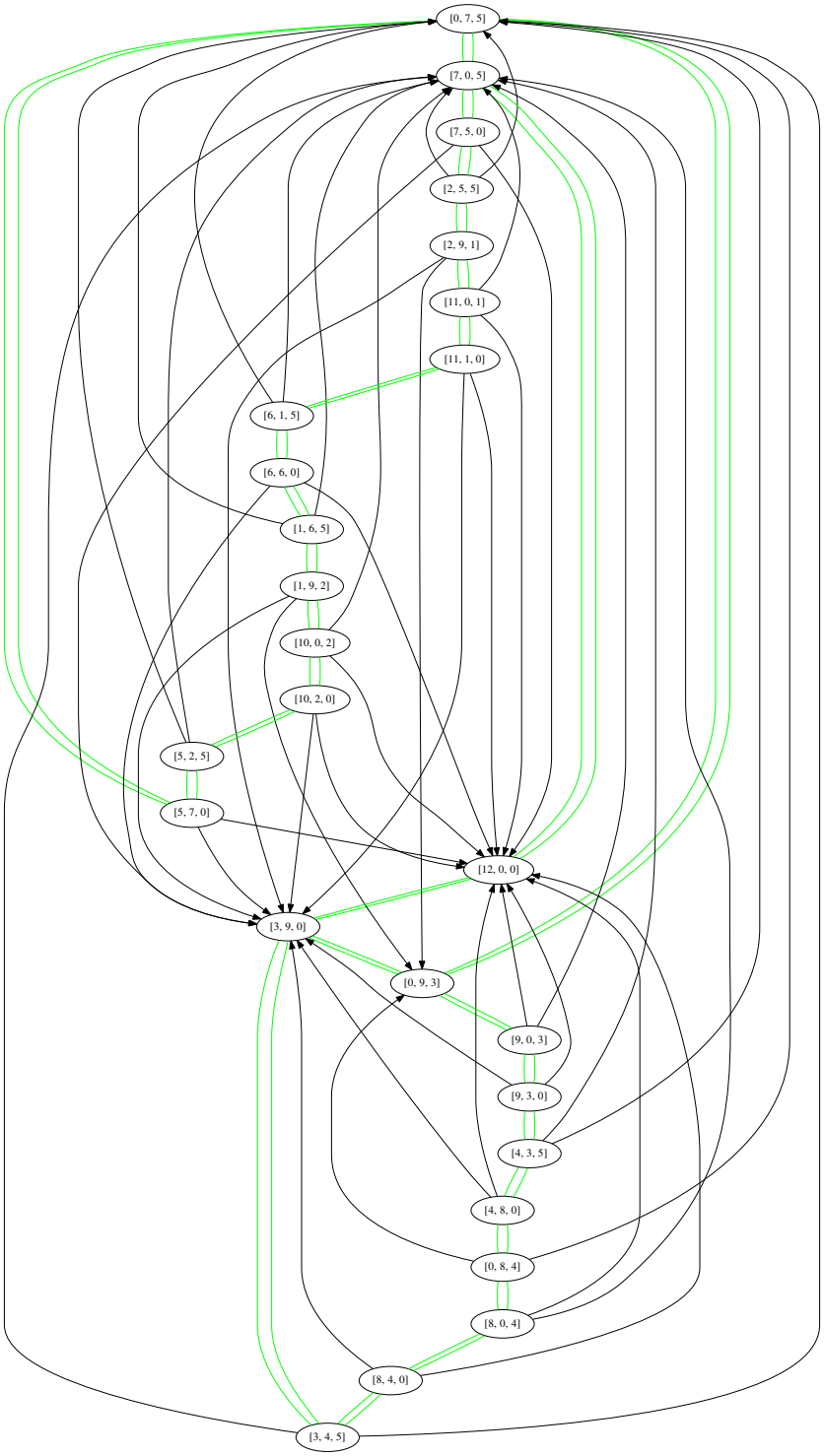

FIGURE 5. The states solving the problem with three jugs of capacities (12,9,5) and dividing 12 units into half (which can be done in 8 steps by walking from [12,0,0] to [6,6,0]). Unreachable vertices (with zero indegree) are not drawn.

4. Overview of the C++ Program

4.1. **Compilation.** The entire program that generates the diagrams and results presented above is shown in the Appendix. The main executable is compiled with

```
make
```

or more explicitly with

```
gcc -O2 -o JugPuzzle JugPuzzle.cxx Jug.cxx JugState.cxx JugGraph.cxx
```

or with

```
autoreconf -i
configure --prefix=‘pwd‘
make
```

4.2. **Use.** The program is called as

`JugPuzzle` *[-a]*

to solve puzzles with known input and output states minimizing the number of steps, or as

`JugPuzzle` *[-a]* `-l` *units*

to solve puzzles with known input states, but output states more vaguely defined to have *units* in at least one jar in the final state, or as

`JugPuzzle -D` *indegree*

to generate `graphviz` source programs for the state diagrams and tables of distances between them.

(1) In the first format, the initial state is specified by the first line of the standard input, and the final state is specified by the second line of the standard input. The states are a list of integers, separated by blanks, which provide the capacity of the first jug, a blank, the liquid in the first jug, a blank, the capacity of the second jug, a blank, the liquid in the second jug, a blank, and so on. So there is an even number of integers in both input lines, and half the count of integers specifies how many jugs are involved. Note that the jug capacities in both lines must match and that the total amount of liquid must be the same in both lines.

(2) In the second format, the input state is specified by the first line of the standard input as above. The final state is implicitly defined by keeping the number and capacities of the jars and having *units* of liquid in any of them.

(3) In the third format, the program does not solve a single puzzle but plots the digraph edges for some constant set of jugs and some constant total amount F of liquid. The first line of the standard input has the format as above. The command line option specifies the minimum indegree nodes must have to be put into the state diagram. To incorporate all nodes, `-D 0` should be used for example.

The option `-a` requests that alternative solutions (with a common minimum number of steps) are also created. This increases the stack space requirements a lot and may only be working for small jug numbers and small amounts of liquid.

4.3. **Examples.**

Example 1. *If we wish to investigate of starting with one jug of capacity 8 filled with 8 units, a second empty jug with capacity 5 and a third empty jug with capacity*

3, and reaching a state with the first jug filled with 4, the second jug filled with 4, and the third jug still empty, we call the program with

```
JugPuzzle
8 8 5 0 3 0
8 4 5 4 3 0
```

Example 2. *If we wish to investigate of starting with one full jug of capacity 7, a second empty jug with capacity 4 and a third empty jug with capacity 3, and reaching a state with the first jug filled with 2, the second jug filled with 2, and the third jug filled with 3, we call the program as*

```
JugPuzzle
7 7 4 0 3 0
7 2 4 2 3 3
```

and the output shows the 6 steps needed.

Example 3. *If you wish to measure 6 liters using 4 and 9 liter jugs, add an auxiliary full 13 liter jug according to the recipe of Section 1.3 and call*

```
JugPuzzle
13 13 9 0 4 0
13 7 9 6 4 0
```

which solves the problem in 9 steps (actually in 8 because the [3,6,4] state is reached earlier).

Example 4. *There is a full 19 liter jar, an empty 13 liter jar, and a 7 liter jar with 1 liter in it. Distribute the liquid such that there are 10 liters in the 19 and 13 liter jar each.*

```
JugPuzzle
19 19 13 0 7 1
19 10 13 10 7 0
```

solves this in 15 steps.

Example 5. *A milk man has two full 10 liter jars left and is approached by two costumers with 4 and 5 liter jars who want to have 2 liters milk each* [2].

```
JugPuzzle
10 10 10 10 5 0 4 0
10 10 10 6 5 2 4 2
```

solves this with 9 steps.

Example 6. *I used the program in 2000 in my posting to* http://mathforum.org/kb/message.jspa?messageID=260017 *to trisect a full container of 36 units into containers of 23, 18 and 9 units in 31 steps:*

```
JugPuzzle
36 36 23 0 18 0 9 0
36 12 23 12 18 12 9 0
```

Example 7. *In a variant we can distribute a full container of 21 ounces given three more empty containers with 11, 8 and 5 ounces into* 3×7 *ounces with*

```
JugPuzzle
21 21 11 0 8 0 5 0
21 7 11 7 8 7 5 0
```

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16	27	18	19	20
steps	2	7	8	17	24	43	66	111	176	289	464	755	1218	1975	3192	5169	8360	13531

TABLE 2. The minimum number for the 3 jugs puzzle where the container size are three consecutive Fibonacci numbers and the middle one contains in the final state one unit less than its capacity.

in 11 steps.

Example 8. *Given 5 mugs with capacity 6 (empty), capacity 5 (full), capacity 4 (empty) and capacities 3 and 2 (both full) redistribute such that each mug is filled with 2.*

```
JugPuzzle
6 0 5 5 4 0 3 3 2 2
6 2 5 2 4 2 3 2 2 2
```

does this in 5 steps.

Example 9. *Given 4 jar with 24 units (full) and 13, 13 and 5 units (all 3 empty), measure 12 units in any of the jars:*

```
JugPuzzle -l 12
24 24 13 0 13 0 5 0
```

generates a solution with 11 steps (where one of the two 13 units jars is not used at all).

Example 10. *The standard sizes of the containers 8, 5 and 3 discussed in Section 1 are three consecutive Fibonacci numbers $F(6)$, $F(5)$, and $F(4)$, and the final filling state of the middle (4) one is one less than its capacity. We let the program compute the number of steps where the three capacities are $F(n)$, $F(n+1)$ and $F(n+2)$ where the largest is initially full, the other two initially empty, and where the final state is $F(n+2)-F(n+1)+1$ in the largest, $F(n+1)-1$ in the middle and again 0 in the smallest. The minimum numbers of the steps for that setup are collected in Table 2. It seems if one requires to end up with $F(n+1)-1$ in any of the containers, and no specification of how much liquid remains in the other two containers, that number of steps can be reduced by one. Two examples of this heuristics: (i) The 17 steps in the table refer to jug capcities (21,13,8) and to the steps [21,0,0] →[8,13,0] →[8,5,8] →[16,5,0] →[16,0,5] →[3,13,5] →[3,10,8] →[11,10,0] →[11,2,8] →[19,2,0] →[19,0,2] →[6,13,2] →[6,7,8] →[14,7,0] →[14,0,7] →[1,13,7] →[1,12,8] →[9,12,0] to move 12 into the middle jug and to clear the small jug, but the last step is not needed if the aim is merely to have 12 units in the middle jar. (ii) The 24 steps in the table refer to the capacities (34,21,13) and to the steps [34,0,0] →[21,0,13] →[21,13,0] →[8,13,13] →[8,21,5] →[29,0,5] →[29,5,0] →[16,5,13] →[16,18,0] →[3,18,13] →[3,21,10] →[24,0,10] →[24,10,0] →[11,10,13] →[11,21,2] →[32,0,2] →[32,2,0] →[19,2,13] →[19,15,0] →[6,15,13] →[6,21,7] →[27,0,7] →[27,7,0] →[14,7,13] →[14,20,0] to move 20 into the middle jug and clear the small jug. But only 23 steps are needed if the 20 units may appear in any jug: [34,0,0] →[13,21,0] →[13,8,13] →[26,8,0] →[26,0,8] →[5,21,8] →[5,16,13] →[18,16,0] →[18,3,13] →[31,3,0] →[31,0,3] →[10,21,3] →[10,11,13] →[23,11,0] →[23,0,11] →[2,21,11] →[2,19,13] →[15,19,0] →[15,6,13] →[28,6,0] →[28,0,6] →[7,21,6] →[7,14,13] →[20,14,0].*

Example 11. *To generate the information of Figure 1 call*

```
JugPuzzle -D 0
8 4 5 0 3 0
```

Example 12. *To generate the information of Figure 3 call*

```
JugPuzzle -D 1
8 8 5 0 3 0
```

Example 13. *To generate the information of Table 1 call*

```
JugPuzzle -D 0
8 8 5 0 3 0
```

Appendix A. Computer Listing

A.1. Makefile.

```
JugPuzzle: JugPuzzle.cxx Jug.h Jug.cxx JugState.h JugState.cxx JugGraph.h JugGraph.cxx
    $(CXX) -O2 -o $@ JugPuzzle.cxx Jug.cxx JugState.cxx JugGraph.cxx
```

A.2. Makefile.am.

```
# $Header:$

AUTOMAKE_OTPIONS =
ACLOCAL_AMFLAGS = ${ACLOCAL_FLAGS}

bin_PROGRAMS = JugPuzzle

JugPuzzle_SOURCES = Jug.cxx Jug.h JugGraph.cxx JugGraph.h JugPuzzle.cxx JugState.cxx JugState.h
```

A.3. configure.ac.

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

# $Header:$

AC_PREREQ([2.68])
AC_INIT([JugPuzzle], [1.0], [mathar@mpia.de])
AC_CONFIG_SRCDIR([JugPuzzle.cxx])
AM_INIT_AUTOMAKE([no-define foreign subdir-objects])
AC_CONFIG_HEADERS([config.h])

AC_PROG_CXX
AC_PROG_INSTALL

AC_CHECK_HEADERS([cstdlib unistd.h])

AC_CHECK_FUNCS([atoi])

AC_CONFIG_FILES([Makefile])

AC_OUTPUT
```

A.4. **Main program JugPuzzle.cxx.**

```
#include <unistd.h>
#include <cstdlib>
#include <iostream>

#include "JugGraph.h"

/*!*
* @brief Solve the J jugs puzzle with specified start and end states.
*
* The synposes are:
* JugPuzzle   # solve the specific jugs problem
* JugPuzzle -a  # solve the specific jugs problem and get all solutions
* JugPuzzle -D <indegree> # generate all states in a dot(1) graph with indegree at least as given
* JugPuzzle -l <liq> # solve a general problem; only liquid volume but not specific
* @return 0 if no error occured.
* 1 if the input data are erroneous (inconsistent).
*/
int main(int argc, char *argv[])
{
    bool solvall(false) ;
    bool solliq(false) ;
    bool dot(false) ;
    int liq(-1) ;
    int indeg(0) ;
    /* option character
    */
    char oc ;
    while (  (oc=getopt(argc,argv,"aD:l:")) != -1 )
    {
        switch(oc)
        {
        case 'a' :
            solvall = true ;
            break ;
        case 'D' :
            dot = true ;
            indeg = atoi(optarg) ;
            break ;
        case 'l' :
            solliq = true ;
            liq = atoi(optarg) ;
            break ;
        case '?' :
            std::cerr << "Invalid command line option " << optopt << std::endl ;
            break ;
        }
    }

    if ( dot ==false && solliq == false)
    {
        /* read the initial state from one line of the stdin
```

```
52          */
53          std::string s;
54          getline(std::cin,s) ;
55          JugState src(s) ;
56          src.thisState =0 ;
57          std::cout << src<< std::endl ;
58
59          /* read the final state from the second line of the stdin
60          */
61          getline(std::cin,s) ;
62          JugState dest(s) ;
63          std::cout << dest<< std::endl ;
64
65          if ( ! src.compatible(dest) )
66          {
67              std::cerr << "Incompatible jug capacities or liquid volumes in " << src<< " and " << dest <
68              return 1;
69          }
70
71          JugGraph::solve(src,&dest,-1,true,solvall) ;
72      }
73      else if ( dot )
74      {
75          std::string s;
76          getline(std::cin,s) ;
77          JugState src(s) ;
78          src.dotGraph(indeg) ;
79
80          std::cout << "/*" << std::endl;
81          src.distances() ;
82          std::cout << "*/" << std::endl;
83      }
84      else if ( solliq && liq >= 0)
85      {
86          /* read the initial state from one line of the stdin
87          */
88          std::string s;
89          getline(std::cin,s) ;
90          JugState src(s) ;
91          src.thisState =0 ;
92          std::cout << src<< std::endl ;
93
94          JugGraph::solve(src,0,liq,true,solvall) ;
95      }
96
97      return 0 ;
98  }
```

A.5. **Jug.h.**

```
1   #pragma once
2
3   #include <iostream>
4   #include <cstdlib>
```

```
5
6   class Jug
7   {
8   public:
9       /** maximum liquid capacity
10      */
11      int capac ;
12
13      /** current amount of liquid
14      */
15      int fill ;
16
17      Jug() ;
18      Jug(int c, int f) ;
19  protected:
20  private:
21  } ;
22
23  bool operator==(const Jug & left, const Jug & right) ;
24  bool operator!=(const Jug & left, const Jug & right) ;
25  bool operator<(const Jug & left, const Jug & right) ;
26  std::ostream & operator<<(std::ostream & os, const Jug & j) ;
```

A.6. **Jug.cxx.**

```
1   #include <iostream>
2
3   #include "Jug.h"
4
5   /** default ctor.
6   * This creates a Jug with an impossible negative capacity
7   * since 2015-09-13
8   */
9   Jug::Jug() : capac(-1), fill(0)
10  {
11  }
12
13  /** Ctor with given capacity and liquid content.
14  * @param[in] c Capacity
15  * @param[in] f liquid content
16  * since 2015-09-13
17  */
18  Jug::Jug(const int c, const int f) : capac(c), fill(f)
19  {
20  }
21
22  /** A sorting operator on two jugs.
23  * @param left The jug on the left hand side of the equal sign.
24  * @param right The jug on the right hand side of the equal sign.
25  * @return True if both jugs have equal capacity and filling.
26  */
27  bool operator==(const Jug & left, const Jug & right)
28  {
29      return ( left.capac == right.capac & left.fill == right.fill) ;
```

```
}

/** A sorting operator on two jugs.
* @param left The jug on the left hand side of the equal sign.
* @param right The jug on the right hand side of the equal sign.
* @return True both jugs have unequal capacity or unequal filling.
*/
bool operator!=(const Jug & left, const Jug & right)
{
    return ! ( left == right ) ;
}

/** A sorting operator on two jugs.
* @param left The jug on the left hand side of the equal sign.
* @param right The jug on the right hand side of the equal sign.
* @return True if the left jug has smaller capacity or smaller filling.
*/
bool operator<(const Jug & left, const Jug & right)
{
    if ( left.capac < right.capac)
        return true;
    else if ( left.capac > right.capac)
        return false;
    else if ( left.fill < right.fill)
        return true;
    else
        /* this here includes the case of equality
        */
        return false;
}

/*!*************************************
* Print the jug's state as its capacity, an underscore, and the current amount of liquid.
* @param[in,out] os The output stream which receivces the string.
* @param[in] j The jug to be printed.
* @return The output stream with the jug's stated printed.
*/
std::ostream & operator<<(std::ostream & os, const Jug & j)
{
        os << " " << j.capac << "_" << j.fill ;
        return os ;
} /* operator<< */
```

A.7. **JugState.h.**

```
#pragma once

/*!***********************************************
* @file
* The interface definition of the JugState class.
*/

#include <iostream>

```

```
10 #include <vector>
11
12 #include "Jug.h"
13
14 /*!***********************************************
15 * @brief A JugState is a snapshot of a list of jugs in some fixed states of liquid volumes.
16 */
17 class JugState
18 {
19 public:
20     /** The collection of partially filled jugs in the current state
21     */
22     std::vector<Jug> jugs ;
23
24     /** If this has been instatiated by a pouring step: the previous state
25     */
26     int prevState ;
27     /** The index of this state in some larger vector. Compatible with prevState.
28     */
29     int thisState ;
30
31     JugState() ;
32     JugState(const int parent) ;
33     JugState(const std::string & bsv) ;
34
35     int F() const ;
36     int J() const ;
37     void addJug(const Jug & j) ;
38     bool hasFill(const int f) const ;
39     int inDegree(const std::vector<JugState>& sts) const ;
40
41     JugState pour(const int src, const int dest, const int parent) const;
42
43     std::vector<JugState> anyPour() const ;
44     std::vector<JugState> allStates() const ;
45     std::vector<JugState> allStates(int remainf, int lowj) const ;
46     void dotGraph(const int indeg=0) const ;
47     void distances() const ;
48
49     bool reversible(const JugState & oth) const ;
50     bool compatible(const JugState & oth) const ;
51 protected:
52 private:
53 } ;
54
55 std::ostream & operator<<(std::ostream & os, const JugState & j) ;
56
57 bool operator==(const JugState & left, const JugState & right) ;
58 bool operator!=(const JugState & left, const JugState & right) ;
59 bool operator<(const JugState & left, const JugState & right) ;
```

A.8. **JugState.cxx.**

```
1  #include <cmath>
2  #include <iostream>
3  #include <sstream>
4
5  #include "JugGraph.h"
6
7  /** Default ctor.
8  * There are no jugs present here
9  * @since 2015-09-13
10 */
11 JugState::JugState() : jugs(), prevState(-1), thisState(-1)
12 {
13 } /* ctor */
14
15 /**
16 * There are no jugs present here
17 * @parent The state that created this state by a pouring step.
18 * @since 2015-09-13
19 */
20 JugState::JugState(const int parent) : jugs(), prevState(parent), thisState(-1)
21 {
22 } /* ctor */
23
24 /** Define a state represented by a vector of integers separated by blanks.
25 * @param bsv The blank separated list of capacities and fillin states.
26 *  The format is
27 *  capac[0] fill[0] capac[1] fill[1] capac[2] fil[2] ...
28 *  which indicates an even number of nonnegative values, 0<=fill[i]<=capac[i]
29 * @since 2015-09-13
30 */
31 JugState::JugState(const std::string & bsv) : jugs(), prevState(-1), thisState(-1)
32 {
33     std::istringstream s(bsv) ;
34     for(;s;)
35     {
36         int cap,f ;
37         s >> cap >> f ;
38         /* exit if the inputs are incoherent
39         */
40         if ( cap <= 0 || f < 0 || f > cap)
41         {
42             std::cerr << "Invalid mix of capacity " << cap << " and liquid volume " << f << std::endl ;
43             exit(EXIT_FAILURE) ;
44         }
45
46         Jug j(cap,f) ;
47         jugs.push_back(j) ;
48         if ( s.eof() )
49             break;
50     }
51 } /* ctor */
52
53 /** Join one jug to the set of jugs
```

```
54  * @param j The additional jug to be added to the set.
55  * @since 2015-09-13
56  */
57  void JugState::addJug(const Jug &j)
58  {
59      jugs.push_back(j) ;
60  } /* ctor */
61
62  /** Compute total liquid content.
63  * @return The sum of all liquid units in all jugs.
64  * @since 2015-09-14
65  */
66  int JugState::F() const
67  {
68      int totf =0 ;
69      for(int j=0 ; j < jugs.size() ; j++)
70          totf += jugs[j].fill ;
71      return totf ;
72  } /* F */
73
74  /** Compute number of jugs.
75  * @return The count of jugs in the set.
76  * @since 2015-09-14
77  */
78  int JugState::J() const
79  {
80      return jugs.size() ;
81  } /* J */
82
83  /** Detect whether any of the jugs has a given filling state.
84  * @param[in] f The amount of liquid searched for.
85  * @return true if one of the jugs has liquid content f;
86  * @since 2015-09-14
87  */
88  bool JugState::hasFill(const int f) const
89  {
90      for(int j=0 ; j <jugs.size() ; j++)
91          if ( jugs[j].fill == f)
92              return true;
93      return false ;
94  } /* hasFill */
95
96  /** Check whether pouring into another state is a reversible step
97  * @param oth The destinate state.
98  * @return true if the oth state can be reached and poured back.
99  */
100 bool JugState::reversible(const JugState & oth) const
101 {
102     if ( J() != oth.J() )
103         return false;
104
105     /* determine source and destination jug indices
106     */
```

```
107     int s = -1 ;
108     int d = -1 ;
109     for(int j=0 ; j < J() ; j++)
110     {
111         if ( jugs[j].capac != oth.jugs[j].capac)
112             return false;
113         const int t = oth.jugs[j].fill - jugs[j].fill ;
114         if ( t > 0 )
115         {
116             if ( d < 0)
117                 d = j;
118             else
119                 return false;
120         }
121         else if ( t < 0 )
122         {
123             if ( s < 0)
124                 s = j;
125             else
126                 return false;
127         }
128     }
129
130     /* we found exactly two jugs that changed their liquid volume
131     * with compatible signs, and either the source jug is full or the
132     * destination jug is empty
133     */
134     if ( s >=0 && d >=0 && (jugs[s].fill == jugs[s].capac) || (jugs[d].fill ==0) )
135         return true;
136     else
137         return false;
138 } /* reversible */
139
140 /** Check whether two states are compatible.
141 * @param oth The state this is to be compared with.
142 * @return true if both states have the same jug sizes and the same total liquid volume.
143 *  This means that from the outside some path through the state
144 *  graph might exist to reach this one from oth or vice versa.
145 */
146 bool JugState::compatible(const JugState & oth) const
147 {
148     /* incompatible if jug numbers or total volume differ
149     */
150     if ( J() != oth.J() || F() != oth.F() )
151         return false;
152
153     for(int j=0 ; j < J() ; j++)
154     {
155         /* incompatible if any pair of jug sizes differ
156         */
157         if ( jugs[j].capac != oth.jugs[j].capac)
158             return false;
159     }
```

```
160     return true;
161 } /* compatible */
162 
163 /** Pour from this jug to another
164 * @param src the 0-based index of the source jug
165 * @param dest the 0-based index of the destination jug
166 * @return The set of the state after pouring.
167 *   This is a state without jugs if the pouring cannot be done according
168 *   to the rules.
169 */
170 JugState JugState::pour(const int src, const int dest, const int parent) const
171 {
172     //JugState newst(this) ;
173     JugState newst(parent) ;
174     /* J is the number of jugs present
175     */
176     const int J = jugs.size() ;
177     /* ensure that the two indices point to existing jugs
178     * and that there is no pouring within a single jug
179     */
180     if ( src >=0 && src < J && dest >=0 && dest < J && src != dest)
181     {
182         /* transferred liquid. Either all of the source jug
183         * or the residual air in the destination jug.
184         */
185         const int t = std::min( jugs[src].fill, jugs[dest].capac -jugs[dest].fill) ;
186         if (t >0 )
187         {
188             /* a useful new state: transfer of nonzero amount
189             */
190             for(int j=0 ; j < J ; j++)
191             {
192                 if ( j == src)
193                 {
194                     Jug newsrc(jugs[j].capac, jugs[j].fill -t) ;
195                     newst.addJug(newsrc) ;
196                 }
197                 else if ( j == dest)
198                 {
199                     Jug newdest(jugs[j].capac, jugs[j].fill +t) ;
200                     newst.addJug(newdest) ;
201                 }
202                 else
203                     /* copy jug unchanged
204                     */
205                     newst.addJug( jugs[j]) ;
206             }
207         }
208     }
209     return newst ;
210 } /* JugState::pour */
211 
212 /** Collect all possible outcomes of a single pouring step.
```

```
213 * @return A list of all subsequence states of pouring once from any of the jugs to any other.
214 * @since 2015-09-13
215 */
216 std::vector<JugState> JugState::anyPour() const
217 {
218     /* the collection of all new states
219     */
220     std::vector<JugState> nextst ;
221     const int J = jugs.size() ;
222     /* any pair of source jug index src and and destination jug index dest.
223     */
224     for(int src =0 ; src < J; src++)
225     {
226         for(int dest =0 ; dest < J; dest++)
227             /* cannot pour a jug into itself */
228             if ( src != dest)
229             {
230                 const JugState resul = pour(src,dest,thisState) ;
231                 if ( resul.jugs.size() > 0 )
232                     /* if this was a valid combination of source and destination index
233                     */
234                     nextst.push_back(resul) ;
235             }
236     }
237     return nextst ;
238 } /* anyPour */
239
240 /** Determine the indegree amongst a set of other states.
241 * @param[in] sts The collection of states.
242 * @return The integer number, >=0, of how many of the sts states can reach this in one step.
243 * @since 2015-09-16
244 */
245 int JugState::inDegree(const std::vector<JugState> & sts) const
246 {
247     int deg =0 ;
248     for(int s = 0 ; s < sts.size() ;s++)
249     {
250         const std::vector<JugState> stsres = sts[s].anyPour() ;
251         for(int j=0 ; j < stsres.size() ; j++)
252         {
253             if ( stsres[j] == *this)
254             {
255                 deg++ ;
256                 break;
257             }
258         }
259     }
260     return deg ;
261 } /* inDegree */
262
263 /** Collect all possible states which have the sam jug sizes and same amount of liquid.
264 * @return A list of all states compatible with the current one.
265 * @since 2015-09-14
```

```
266 */
267 std::vector<JugState> JugState::allStates() const
268 {
269     /* the available total liquid
270     */
271     const int totf = F() ;
272     return allStates(totf,0) ;
273 } /* allStates */
274
275 /** Collect all possible states which have the sam jug sizes and same amount of liquid.
276 * @param[in] remainf The total amount of liquid to be distributed in the jugs of higher index.
277 * @param[in] lowq The lowest index of the jugs yet to be filled.
278 * @return A list of all states compatible with the current one.
279 * @since 2015-09-14
280 */
281 std::vector<JugState> JugState::allStates(int remainf, int lowq) const
282 {
283     /* empty number of states at start
284     */
285     std::vector<JugState> sts;
286     /* copy the characteristics (size and filling) of the jugs already fixed in content
287     * into a pivotal jug
288     */
289     JugState piv = *this ;
290     piv.prevState = -1 ;
291
292     if ( lowq == J()-1 )
293     {
294         /* last jug to be filled; which means all liquid must fit in there
295         */
296         if ( remainf <= jugs[lowq].capac)
297         {
298             piv.jugs[lowq].fill = remainf ;
299             sts.push_back(piv) ;
300         }
301     }
302     else if ( lowq < J()-1 )
303     {
304         /* more jugs to be filled. Recursive approach by filling number lowq
305         * and the higher ones with the residual liquid.
306         */
307         for(int f=0; f <= piv.jugs[lowq].capac && f <= remainf ; f++)
308         {
309             piv.jugs[lowq].fill = f ;
310             const std::vector<JugState> recv = piv.allStates(remainf-f, lowq+1) ;
311             for(int j=0 ; j < recv.size() ; j++)
312                 sts.push_back(recv[j]) ;
313         }
314     }
315
316     return sts;
317
318 } /* allStates */
```

```
319
320 /** Generate all compatible states in the dot(1) graphviz format
321 * @param[in] indeg The minimum indegree of states to be incorporated
322 */
323 void JugState::dotGraph(const int indeg) const
324 {
325     /* generate the states with the same amount of liquid
326     */
327     const std::vector<JugState> sts = allStates() ;
328     std::cout << "/* " << sts.size() << " states, indegree " << indeg << " */ \n" ;
329     std::cout << "digraph L {\n" ;
330     /* generate for each state all reachable states with one step
331     */
332     for(int s=0 ; s < sts.size() ; s++)
333     {
334         const int degS = sts[s].inDegree(sts) ;
335         if ( degS >= indeg)
336         {
337             const std::vector<JugState> dest = sts[s].anyPour() ;
338             for(int d=0 ; d < dest.size() ; d++)
339             {
340                 const int degD = dest[d].inDegree(sts) ;
341                 if ( degD >= indeg)
342                 {
343                     std::cout << "\"" << sts[s] << "\" -> \"" << dest[d] << "\"" ;
344                     if (sts[s].reversible(dest[d]) )
345                         std::cout << " [color = \"green\" dir=\"none\" ]" ;
346                     std::cout << " ; /* " << degS << " " << degD << " */\n" ;
347                 }
348             }
349         }
350     }
351
352     std::cout << "}\n" ;
353 } /* dotGraph */
354
355 /** Generate a Table of the distances (shortest steps) in the compatible states.
356 * The table is printed in LaTex style to stdout.
357 */
358 void JugState::distances() const
359 {
360     /* generate the states with the same amount of liquid
361     */
362     const std::vector<JugState> sts = allStates() ;
363     std::cout << "\\begin{tabular}{" ;
364     for(int s=0 ; s <= sts.size() ; s++)
365         std::cout << "r" ;
366     std::cout << "}\n" ;
367
368     for(int s=0 ; s < sts.size() ; s++)
369         std::cout << " & \\rotatebox{90}{$" << sts[s] << "$}" ;
370     std::cout << "\\\\\n" ;
371
```

```
372     std::cout << "\\\\\n\\hline\n" ;
373     /* generate for each state all reachable states with one step
374     */
375     for(int s=0 ; s < sts.size() ; s++)
376     {
377         std::cout << "$" << sts[s] << "$"  ;
378         for(int d=0 ; d < sts.size() ; d++)
379         {
380             if ( s==d)
381                 std::cout << " & 0 " ;
382             else
383             {
384                 int stps = JugGraph::solve(sts[s], & sts[d], -1, false, false) ;
385                 std::cout << " & " << stps ;
386             }
387         }
388         std::cout << "\\\\\n" ;
389     }
390
391 } /* distances */
392
393 /** Print the filling levels into an ASCII Stream
394 * The jugs are represented as a bracketed and comma-seperated list of integers,
395 * which denote their individual liquid contents.
396 * @param os The stream to add the state
397 * @param jst The jug set to be printed.
398 * @return The stream after printing.
399 * @since 2015-09-13
400 */
401 std::ostream & operator<<(std::ostream & os, const JugState & jst)
402 {
403     os << "[" ;
404     for(int j=0 ; j < (int) jst.jugs.size() ; j++)
405     {
406         if ( j > 0)
407             os <<"," ;
408         os << jst.jugs[j].fill ;
409     }
410     os << "]" ;
411     return os ;
412 } /* operator<< */
413
414 /** A sorting operator on two jug sets.
415 * @param left The jug set on the left hand side of the equal sign.
416 * @param right The jug set on the right hand side of the equal sign.
417 * @return True If all jugs have equal capacity and equal filling.
418 * @since 2015-09-13
419 */
420 bool operator==(const JugState & left, const JugState & right)
421 {
422     if ( left.jugs.size() != right.jugs.size() )
423         return false;
424
```

```
425     for(int j=0 ; j < left.jugs.size() ; j++)
426         if ( left.jugs[j] != right.jugs[j])
427             return false ;
428 
429     return true;
430 }
431 
432 /** A sorting operator on two jug sets.
433 * @param left The jug set on the left hand side of the unequal sign.
434 * @param right The jug set on the right hand side of the unequal sign.
435 * @return True If some jugs have different capacity or different filling.
436 * @since 2015-09-13
437 */
438 bool operator!=(const JugState & left, const JugState & right)
439 {
440     return ! (left == right) ;
441 }
```

A.9. **JugGraph.h.**

```
1  #pragma once
2  
3  #include <iostream>
4  
5  #include <vector>
6  
7  #include "JugState.h"
8  
9  class JugGraph
10 {
11 public:
12     /** A basically unordered list of vertices in the state diagram
13     */
14     std::vector<JugState> vertices ;
15 
16     /** A marker in the vertices which separates the parent states
17     * and the current generation
18     */
19     int Nparent ;
20 
21     JugGraph() ;
22     JugGraph(JugState root) ;
23     void addState(const JugState & st) ;
24 
25     void newGenerat(const bool multiRoute) ;
26     static int solve(const JugState & src, const JugState * destin,  const int destfil, const bool verb
27 
28 protected:
29 private:
30 } ;
31 
32 // std::ostream & operator<<(std::ostream & os, const JugState & j) ;
```

A.10. **JugGraph.cxx.**

```
#include <cmath>

#include "JugGraph.h"

/** Default ctor.
* There are no jugs present here
* @since 2015-09-13
*/
JugGraph::JugGraph() : vertices(), Nparent(0)
{
} /* ctor */

/** ctor which pushes one JugState at the tree top.
* @param st The state at the top of the tree of pouring liquid.
* @since 2015-09-14
*/
JugGraph::JugGraph(JugState root) : vertices(), Nparent(0)
{
    // root.prevState = 0 ;
    root.prevState = -1 ;
    vertices.push_back(root) ;
} /* ctor */

/** Joint one jug to the set of jugs
* @param j The additional jug to be added to the set.
* @since 2015-09-13
*/
void JugGraph::addState(const JugState &st)
{
    vertices.push_back(st) ;
} /* addState */

/** add a new generation of states to the existing ones.
* @param multiRoute If true generate paths where states with different parents are considered different
* @since 2015-09-13
*/
void JugGraph::newGenerat(const bool multiRoute)
{
    /* we add the states that can be generted from the current generation,
    * which includes the vertices above the generation marker
    */
    const int oldNvertic = vertices.size() ;
    for(int v=Nparent ; v < oldNvertic ; v++)
    {
        /* create any new states arising from the v'th state
        */
        std::vector<JugState> vs = vertices[v].anyPour() ;
        /* add all of those not yet present in older generations
        * At this point there is some loss of information because we
        * do not log states that are equal but arrive through different parents.
        */
        for(int s =0 ; s < (int) vs.size() ; s++)
        {
```

```
54              /* candidate state
55              */
56              JugState & candi = vs[s] ;
57              candi.thisState = vertices.size() ;
58              /* candidate known in older generations?
59              */
60              bool known =false ;
61              for(int o=0 ; o < Nparent && !known ;o++)
62                  if ( vertices[o] == candi)
63                      known =true ;
64
65              if ( ! known)
66              {
67                  if ( multiRoute)
68                      addState(candi) ;
69                  else
70                  {
71                      /* put only a single candidate into the new
72                      * generation. I.e., search also the current generation for siblings
73                      */
74                      known = false;
75                      for(int o=Nparent ; o < (int) vertices.size() && !known ;o++)
76                          if ( vertices[o] == candi)
77                              known =true ;
78
79                      if ( ! known)
80                          addState(candi) ;
81                  }
82              }
83          }
84      }
85
86      /* update generation marker
87      */
88      Nparent = oldNvertic ;
89  } /* newGenerat */
90
91  /*!*********************************
92  * Solve a puzzle given the start state and either the fully qualified or general final state.
93  * @param[in] src The state at the start of the pouring steps.
94  * @param[in] destin The state at the end of the pouring steps.
95  *   This should be set to 0 if the target state is characterized by the destfil
96  *   value and not by a precise set of filling state of all jugs.
97  * @param[in] destfil The amount of liquid in at least one of the jugs at the end of the steps.
98  *   This should be set to a negative value to indicate that this type of generalized
99  *   search should not be attempted.
100 * @param[in] verb Print the pouring steps if a solution is found.
101 * @param[in] multiRoute Print more than one way of obtaining the destin state if possible.
102 * @return The number of steps needed. -1 if the destin state cannot be reached.
103 */
104 int JugGraph::solve(const JugState & src, const JugState * destin, const int destfil,
105     const bool verb, const bool multiRoute)
106 {
```

```
107     int steps(0) ;
108     bool solved(false);
109     JugGraph gens(src) ;
110
111     if ( destin == 0 || src != * destin)
112     {
113         /* start a loop which at each round adds the
114         * reachable states from the previous loops to the gens graph.
115         */
116         do {
117             steps++ ;
118             /* add all the nodes that are reachable from the
119             * nodes that were generated by the previous loop, but not
120             * those that are already in graph (i.e., avoiding walking
121             * around loops in the graph.
122             */
123             gens.newGenerat(multiRoute) ;
124
125             /* no new elements generated: unreachable final state.
126             */
127             if ( gens.Nparent == gens.vertices.size() )
128             {
129                 if (verb)
130                     std::cout << "no solution" << std::endl ;
131                 return -1 ;
132             }
133
134             /* check if any of the new elements are equal to destin
135             */
136             solved = false;
137
138             for(int s = gens.Nparent ; s < gens.vertices.size() ; s++)
139             {
140                 if ( destfil >=0 && gens.vertices[s].hasFill(destfil)
141                     || destin &&  gens.vertices[s] == *destin)
142                 {
143                     solved =true ;
144
145                     if (verb)
146                     {
147                         /* print the stack of JugStates tracking backwards
148                         * through the history of the generations
149                         */
150                         std::cout << steps << " steps " << std::endl ;
151                         int histo = s ;
152                         for(;;)
153                         {
154                             std::cout << gens.vertices[histo] << std::endl ;
155                             histo = gens.vertices[histo].prevState ;
156                             if ( histo<0 )
157                                 break;
158                         }
159                     }
```

```
                    if ( !multiRoute)
                        return steps ;
                }
            }

        } while (!solved) ;
    }

    return steps ;
} /* solve */
```

References

1. Hassan Aref, *Trilinear coordinates in fluid mechanics*, pp. 568–581, Springer, 1992.
2. Johnny Ball, *Ball of confusions: puzzles, problems and perplexing posers*, Icon books, 2011.
3. H. S. M. Coxeter and Samuel L. Greitzer, *Geometry revisited*, Mathem. Assoc. Am., 1967.
4. Zdravko Voutov Lalchev, Margarita Genova Varbanova, and Irirna Zdravkova Voutova, *Perelman's geometric method of solving liquid pouring problems*, Proc. 6th Mediterranean Conf. Math. Education (Plovdiv, Bulgaria), 22–26 April 2009.
5. M. A. Murray-Lasso, *Math puzzles, powerful ideas, algorithms and computers in teaching problem-solving*, J. Appl. Res. Techn. **1** (2003), no. 3, 215–234.
6. Marilyn A. Reba and Douglas R. Shier, *Puzzles, paradoxes and problem solving*, CRC press, 2014.
7. Martin Rem and Young il Choo, *A fixed-space program of linear output complexity for the problem of the three vessels*, Sci. Comput. Program. **2** (1982), no. 2, 133–141.
8. J. P. Saksena, *Stochastic optimal routing*, Unternehmensforschung **12** (1968), no. 1, 173–177.
9. Shai Simonson, *Public key cryptography*, MAA Notes, vol. 68, 2005.
10. ———, *Rediscovering mathematics: You do the math*, Math. Assoc. Am., 2011.
11. Glânffrwd P. Thomas, *The water jugs problem: solutions from artificial intelligence and mathematical viewpoints*, Math. School **24** (1995), no. 5, 34–37.
12. M. C. K. Tweedie, *A graphical method of solving tartaglian measuring puzzles*, Math. Gaz. **23** (1939), no. 255, 278–282.

URL: `http://www.mpia.de/~mathar`

Max-Planck Institute of Astronomy, Königstuhl 17, 69117 Heidelberg, Germany