MATH PUZZLES, POWERFUL IDEAS, ALGORITHMS AND COMPUTERS IN TEACHING PROBLEM-SOLVING

M.A. Murray-Lasso

Computer-Aided Teaching Unit, Department of Systems Engineering, Graduate School of Engineering, National University of Mexico, Ciudad Universitaria, Mexico City, D.F., 04510, MEXICO

Received June 13th 2001 and accepted October 29th 2002

ABSTRACT

With the current interest in teaching problem-solving, it is important that the example problems that are presented to students have important educational value. Many mathematical puzzles, while requiring little specific background to be posed and remembered, are paradigms of important practical problems. If the teacher chooses the right puzzles, he/she not only will have motivated the student's interest, but will also have the opportunity of teaching powerful ideas that can be applied in a multitude of similar problems. Moreover the teacher will also be able to teach the corresponding solution algorithms and the way computers can be used as tools in problem-solving. The article presents these topics through an extensive example based on a well-known, apparently simple, wine-pouring puzzle that is intimately related to the concepts of state, shortest-routes and many management, engineering and industrial problems.

KEYWORDS: Puzzles, Powerful Ideas, Algorithms, Computers, Problem-Solving, Paradigms, Graphs, States, Recursion, Shortest-Routes.

1. INTRODUCTION

The author's 42 years of teaching experience has led him to the conclusion that to acquire problem-solving skills it is necessary for the students to solve many different types of problems under the guidance of a wise and experienced teacher. Just as 20 year's experience built by repeating the same chores year after year is only worth one year's experience, the quality and variety of problems solved by students is crucial. Nothing is gained by solving countless problems in which all the student has to do is replace variables in a formula by numbers; a few such problems suffice. What is necessary is for the teacher to collect good problems that illustrate situations that appear in many settings and to discuss such problems in great detail, including the different applications, the theories and powerful ideas that can be applied to their solution, the algorithms and associated data structures and the use of the computer as a tool for solving problems. Paradigmatic problems such as the second order system in the study of dynamics allow the teacher to introduce generally useful concepts, such as frequency response, natural frequencies, resonance, dynamic stability, and others. Teachers should constantly look for such rich paradigms.

A good source of paradigms that have many practical interpretations are mathematical puzzles. Games and puzzles have been the source of several branches of mathematics such as Graph Theory and Probability Theory. The first was founded by Euler motivated by the Seven Bridges of Königsberg Puzzle [1], while the second was founded by Pascal motivated by discussions about games of chance with Fermat [2 - 3]. According to Leibnitz, quoted in [3], "...the games by themsalves are worthy to be studied and if some penetrating mathematician meditated upon them he

would find many important results, for man has never shown as much ingenuity as in his plays." Recently, a good amount of research on Artificial Intelligence was done by studying the possibilities that computers play well games such as chess and checkers [4]. In Operations Research many important applications are prototyped by models and methods that receive colorful names such as "The Knapsack Problem," "The Traveling Saleman Problem," The Marriage Problem," "The Monte Carlo Method," "The Chinese Postman," and so on [5].

Although most people solve puzzles by trial and error, the solution to many of them can be formalized and theories built to assure that a solution for a whole family of problems can be found algorithmically. In the building of such theories, powerful ideas, well worth the student's time spent in learning and applying them, often arise.

In this paper some of these ideas are illustrated with a wine-pouring puzzle [6]. Powerful ideas such as that of a "state," "graph," (in the sense of Euler's Graph Theory), "recursion," "labeling," and excellent algorithms such as Dijkstra's Shortest Route Algorithm are introduced. Connections between the wine-pouring problem and the shortest-route problem and some of its interpretations and applications are explored. Finally, the way to use a computer to implement the algorithms in Logo and BASIC is illustrated and program listings are included. Many problems similar to the one given here exist and can receive a similar treatment. Papers detailing them for teachers would be a welcome addition to the problem-solving and educational-computing literature.

2. A WINE-POURING PUZZLE

Two people want to divide evenly the wine contained in a full 8 gallon jug. They have no liquid-measuring device and the jug has no markings. However, available to them are two additional unmarked empty jugs with capacities 5 and 3 gallons. How can they measure 4 gallons?

This well-known puzzle is usually solved by trial and error either experimentally or on paper by trying ideas until one hits on the solution. To appreciate the advantages of having an organized method of solution that always works, the reader should stop reading and attempt finding a solution. The problem is relatively easy and should take no more than a few minutes to solve.

3. STATES

The first powerful idea that will be introduced will be the idea of the state of a system.The state concept is very important in applied mathematics and central in computer science as it appears in such basic concepts as finite-state machines including Turing Machines and algorithms. We will not give a formal definition of state but rather will illustrate the idea in the context of the puzzle at hand. If the reader has either solved the problem or at least seriously attempted a solution, it should be clear by now that the four gallons could be measured as a result of pourings between the jugs. Since none of the jugs are marked, the only meaningful situations are those in which, taking a jug containing some wine, pourings are made to other jugs until either the jug being poured into is full, or the jug from which the pouring is done is empty. Intermediate situations are indeterminate. We can describe the condition or state of the system by means of three ordered numbers indicating the amount of wine in each of the 8, 5 and 3 gallon jugs. The initial state is given by (8, 0, 0). There are only two states that can be reached in one pouring from the initial state. To arrive at the first one the wine in the 8 gallon jug is poured into the 5 gallon jug until it fills up. The new state is (3, 5, 0). The only other possibility is to pour the wine from the 8 gallon jug into the 3 gallon jug until this last jug fills up. The corresponding new state is (5, 0, 3).

Notice that since we will assume no wine is spilled, the sum of the numbers representing a state must always be 8. This means that we can describe the state by any two of the three numbers and deduce the third number by subtraction. We will however, for the sake of clarity keep the 3 number description.

4. GRAPHS

A second powerful idea that arises in the solution to the problem at hand is that of a graph. This is a concept different from the familiar graphs used to represent functions such as the growth of the National Debt with time. The graphs

we are talking about are collections of points and lines connecting the points. The points are often referred to as vertices or nodes and the lines as links or edges. We will associate the states with the points and the operation of pouring with the lines. The graph corresponding to the 3 states above mentioned is shown in Fig. 1.

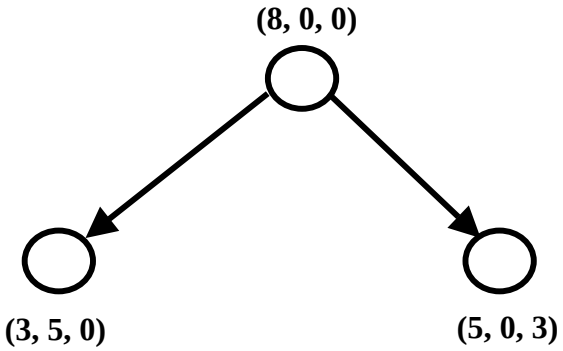

Figure 1.

Notice that the geometric position of the states as well as the form and length of the lines are unimportant, their role is to visualize structural properties rather than represent physical distances and spatial forms. Notice also that the lines or edges have an arrow indicating a traverse direction from state to state. This property gives the edges the name of oriented or directed edges and the graphs the name of oriented or directed graphs. The property is obviously important because, for example, if after several pourings we manage to get the state (4, 4, 0) it is obvious that we can go from there to the original state (8, 0, 0) in one pouring (pour from the 5 gallon jug all the wine into the 8 gallon jug until the 5 gallon jug is empty – which coincides with the event that the 8 gallon jug is filled) while the opposite action is meaningless, otherwise we would be able to solve the puzzle in one pouring.

Lets now continue drawing the graph of Fig. 1. From state (3, 5, 0) we can go in one pouring to the states (8, 0, 0), (0, 5, 3), (3, 2, 3). One of the states is the initial state that had already appeared before, so we draw a line connecting the state (3, 5, 0) directed toward the already-in-place state (8, 0, 0). The other two states generate new states that had not appeared before, so we draw the new states and the corresponding connecting lines with their arrows. (The states that had previously appeared only generate new connecting lines.) We next examine state (5, 0, 3). It can be connected to states (8, 0, 0), (0, 5, 3), (5, 3, 0), one of which had already appeared and two new ones. Next we examine the new states that have appeared in the process and that have not been "expanded" (that is, examined to see what states can be connected to it.) The process can be continued until the graph stops growing because all states that are reachable from the initial state have appeared. We are sure that the graph will eventually stop growing because there is no way any of the states can have in any of its components anything different from an integer between 0 and and the capacity of the jug corresponding to the place in the trio. Hence a bound on the number of states is given by $9\times6\times4 = 216$. That is, the graph can not have any more than 216 nodes. In actual fact, because, among other things, the full capacities can not appear simultaneously in the same state (since they would violate the fact that the component numbers of the state have to add to 8) nor can states like (0, 0, 0), (1, 1, 1), ... appear unless the three numbers add to 8, the graph has only 16 different states. All the states and the transitions between the states appear in the complete state diagram of Fig. 2.

Observing Fig. 2, our original problem can be turned into: Find a route that going in the sense of the arrows starts at state (8, 0, 0) and ends in state (4, 4, 0). The following sequence of states is one such route: (8, 0, 0), (3, 5, 0), (3, 2, 3), (6, 2, 0), (6, 0, 2), (1, 5, 2), (1, 4, 3), (4, 4, 0). The corresponding pourings are: Pour from the 8 gallon jug into the 5 gallon jug until it is filled. We now have in order of descending capacities of the jugs 3, 5 and 0 gallons. Now pour from the 5 gallon jug into the 3 gallon jug until it is filled. We now have 3, 2, 3 gallons. Pour from the 3 gallon jug into the 8 gallon jug until the 3 gallon jug is empty. We now have 6, 2, 0 gallons. Pour from the 5 gallon jug all that it has into the 3 gallon jug. We now have 6, 0, 2 gallons. Pour from the 8 gallon jug into the 5 gallon jug until it is filled. We now have 1, 5, 2 gallons. Pour from the 5 gallon jug into the 3 gallon jug until it is filled. We now have 1, 4, 3 gallons.

Finally pour from the 3 gallon jug all its contents into the 8 gallon jug. We now have 4, 4, 0 gallons and we have solved the problem.

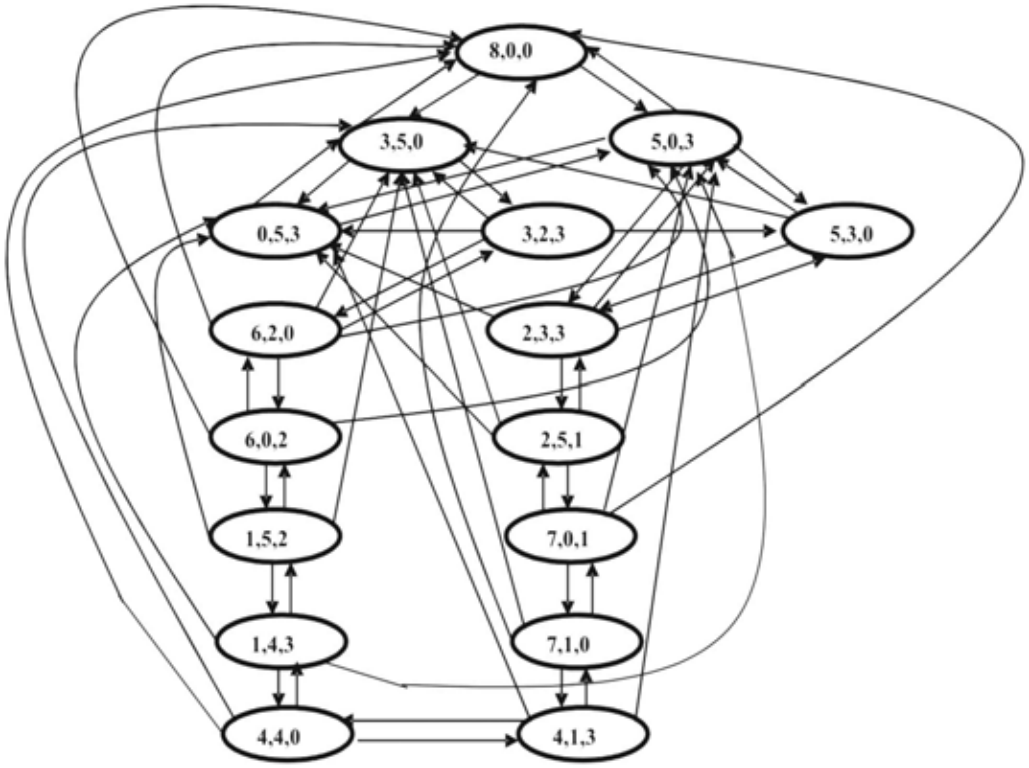

Figure 2.

5. ENTER THE COMPUTER

Building the graph of Fig. 2 can be laborious, and it definitely would be very laborious for analogous puzzles where the capacities of the jugs are larger numbers such as 69, 37 and 32. In such cases we can resort to use a computer to do the work for us. In this introductory presentation we will only go as far as attempting to obtain a structured list of states. This is a reasonable decision since a very large complicated graph is not much of use since it is very easy to loose sight of the connecting lines. We will start with the original state (8,0,0) and list in the same line those states that can be reached in one pouring. Then we process each of these new states starting a line with each one of them and listing in the same line those states which, not having appeared before, can be reached in one pouring. We continue with this process until all the states have been processed and no new states appear. The design of a program to do this is based on the fact that given three numbers *(x,y,z)* representing a state and given three capacities of jugs in descending order *(a,b,c),* we can write a procedure to generate the possible states that can be reached in one pouring. There are up to 4 possibilities for each component of the state: fill one of the other two jugs or empty the full content of the jug from which we are pouring into one of the two other jugs. Although this would imply 12 different possibilities for the 3 components of the state, in practice there are many less transitions from one state to another states because some of the jugs may be empty and no pouring can be done from them or some jugs may be full and no pouring can be done into them. For example, in Fig. 2, none of the states has more than 4 lines leaving, although some have many more lines arriving.

A program titled **STATES** has been written in LogoWriter and its listing appears in Listing 1. It uses the auxiliary procedure **NEW?** Following are some comments regarding these two procedures:

The auxiliary procedure **NEW?** Starts by checking if the list **p** representing a new generated state is already a member of the list of lists **r.** If not, it appends **p** to **r** and to **f**, which are lists external to **NEW?** used for accumulating and processing the states in order of appearance; **NEW?** then appends the distance of **p** from the origin vertex to the external list **t** and outputs TRUE; otherwise, in case **p** has already appeared in list **r**, it outputs FALSE. (See procedure **STATES** below for clarifying the meaning of lists **p, r, t**)

The procedure **STATES** has seven parameters. The first three: ***a, b, c,*** which are held constant throughout the problem, correspond to the capacities of the three jugs in gallons. (For the example given above these are 8, 5 and 3.) The next two parameters are lists of lists whose components are 3-element lists, each representing a state. List **f** is used to process the states generated in order of appearance. List **r** accumulates the states generated so a state can be checked for previous appearances and is output by the procedure. The next parameter, **n**, is a counter that keeps track of the number of times the procedure is invoked and thus numbers the different states generated that can be reached from the starting state. The last parameter, **t**, is a list of distances in edges from the origin vertex, one member for each of the states in the order in which they are numbered by the sixth parameter. **STATES** calls itself recursively and stops and outputs list **r** when list **f** becomes empty. The local variables ***x, y, z*** are used to hold the quantities of wine in the jugs in descending order of their capacities. After the initial **IF** statement to stop the recursion, each **IF** statement starting a paragraph in the listing of **STATES** analizes one of the two possibilities described above for transition to new states (12 possibilities for the 6 paragraphs). Before a new state is appended at the end of lists **f** and **r**, the logical procedure **NEW?** is called to see if the state generated is a new one by searching for it in the list **r**. Only the states that have not previously appeared are **LPUT** at the end of both **r** and **f** as a side effect of **NEW?**

As for the printed output of **STATES,** for each parent node, a trio is printed followed by its distance in edges from the origin vertex. To make the output understandable, one dot is printed before the distance and two dots are printed before the list of children states each of whose members are separated by a dot. The program changes line for each parent state. At the end of the procedure, **STATES** calls itself dropping the trio corresponding to the state just processed from the list **f**, dropping the corresponding distance from the list **t** and increasing by one the counter **n**. Eventually **f** becomes empty and the procedure stops.

```
to new? :p
        ifelse not member? :p :r
[make "r lput :p :r make "f lput :p :f make "t lput (first :t) + 1
          :t op "true]
        [op "false]
end

to states :a :b :c :f :r :n :t
if :f = [] [op :r]
(type "# :n "PARENT= first :f ". first :t ".. )
make "x first first :f
make "y first bf first :f
make "z first bf bf first :f

if :x > 0 [ifelse :x > (:b - :y)
            [if new? (se :x + :y - :b :b :z)
              [(type last :f ". )]]
                [if new? (se 0 :x + :y :z)
              [(type last :f ".)]]]
```

```
if :x > 0 [ifelse :x > (:c - :z)
                [if new? (se :x + :z - :c :y :c)
                  [(type last :f ". )]]
                [if new? (se 0 :y :x + :z)
                  [(type last :f ".)]]]

if :y > 0 [ifelse :y > (:a - :x)
                [if new? (se :a :y + :x - :a :z)
                      [(type last :f ". )]]
                    [if new? (se :x + :y 0 :z)
                      [type last :f ".)]]]

if :y > 0 [ifelse :y > (:c - :z)
                    [if new? (se :x :y + :z - :c :c)
                      [(type last :f ". )]]
                    [if new? (se :x 0 :y + :z)
                      [(type last :f ".)]]]

if :z > 0 [ifelse :z > (:a - :x)
                [if new? (se :a :y :z + :x - :a)
                  [(type last :f ". )]]
                    [if new? (se :x + :z :y 0)
                      [(type last :f ".)]]]

if :z > 0 [ifelse :z > (:b - :y)
                    [if new? (se :x :b :y + :z - :b)
                      [(type last :f ". )]]
                    [if new? (se :x :y + :z 0)
                      [(type last :f ".)]]]
print "
states :a :b :c bf :f :r :n + 1 bf :t
end
```

6. LISTING 1

The most important parts of the program that is shown in Listing 1 are the six groups of instructions (which can be called paragraphs) each starting with lines of the form

"if :v > 0 [ifelse .... "

with v standing for either x, y, or z. We will explain the first of these paragraphs. The others have very similar explanations.

The first paragraph is meant to detect the situation when the largest jug contains some wine, and the intention is to pour from that jug into one of the other jugs. This is tested with a conditional instruction that controls the execution of the instruction between the outermost brackets starting immediately after the 0 and ending with the paragraph. Within the first conditional (that is assuming the largest jug has some wine) the program examines two situations: 1) the amount contained by the largest jug (:x) is larger than whatever capacity is left {(:b - :y)} in the middle jug; 2) the amount contained by the largest jug is equal to, or less than, the capacity left in the middle jug. In the first case, the first event to happen is that the middle jug will be filled, hence the state to be obtained will be: the first component will be whatever the largest jug had minus the capacity that was left in the middle jug, that is :x – (:b - :y) = :x + :y - :b; the second component will be the total capacity of the middle jug, that is :b; the third component will be whatever it

was before the pouring, that is :z. In the second case the first event to happen is that the largest jug will become empty and hence the state to be arrived at will be: the first component will be 0; the second component will be whatever the middle jug had plus whatever the largest jug had, that is :y + :x = :x + :y; the third component remains the same, that is :z.

Nothing will happen if the state to be arrived at had already appeared, a situation which is handled through a third concatenated **IF** instruction with the logical procedure **NEW?**. In case all conditions are met, the program types without changing lines the new states followed by a dot. Similar situations are tested for the other two jugs in the other paragraphs when they contain some wine (:y > 0, :z > 0) which is susceptible of being poured into other jugs. All new states are printed, first as parents followed by their distance to the origin state and by their succeeding neighbors.

The procedure **NEW?** not only tests for previous appearances of a state; as a side effect it appends new states to list **r** where all new states that appear are accumulated and which is output at the end of the execution by the **STATES** procedure and is printed in the command zone via the **SHOW** instruction which the user types. **NEW?** also updates the **f** list which is used to manage the processing of the new states queuing them up and releasing them after they are processed, and the **t** list which manages the distances of the states from the origin state. The release of states already processed is accomplished by a recursive self call by the procedure **STATES** in the next to last line of its listing, in which the arguments of the call drop the first element of both **f** and **t.** The recursive call also takes care of the numbering of the states increasing by one the counter **n** with every call. The third from the end instruction in the listing of the procedure **STATES** changes line in the printing for every new PARENT.

A run of the procedures of Listing 1 to generate all the states reachable from the initial state (8,0,0) of the 8, 5, 3 gallon jug problem follows: (The underlined text is typed by the user, the rest is generated by the programs, the part shown in bold characters appears in the command zone, the normal type appears in the work zone)

SHOW STATES 8 5 3 [[8 0 0]] [[8 0 0]] 1 [0]
1 PARENT= 8 0 0 . 0 .. 3 5 0 .5 0 3 .
2 PARENT= 3 5 0 . 1 .. 0 5 3 .3 2 3 .
3 PARENT= 5 0 3 . 1 .. 5 3 0 .
4 PARENT= 0 5 3 . 2 ..
5 PARENT= 3 2 3 . 2 .. 6 2 0 .
6 PARENT= 5 3 0 . 2 .. 2 3 3 .
7 PARENT= 6 2 0 . 3 .. 6 0 2 .
8 PARENT= 2 3 3 . 3 .. 2 5 1 .
9 PARENT= 6 0 2 . 4 .. 1 5 2 .
10 PARENT= 2 5 1 . 4 .. 7 0 1 .
11 PARENT= 1 5 2 . 5 .. 1 4 3 .
12 PARENT= 7 0 1 . 5 .. 7 1 0 .
13 PARENT= 1 4 3 . 6 .. 4 4 0 .
14 PARENT= 7 1 0 . 6 .. 4 1 3 .
15 PARENT= 4 4 0 . 7 ..
16 PARENT= 4 1 3 . 7 ..

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

7. RUN OF THE PROCEDURES STATES AND NEW?

Note that the list of lists in bold characters corresponds to the trios in the normal type immediately following the PARENTS= labels. From the printed output of STATES we can easily build the tree appearing in Fig. 3, from which the solution of the puzzle can be found. A path from the initial state (8,0,0) to the final state (4,4,0) determines the sequence of pourings. The solution found in this way requires 7 pourings. The number of pourings necessary for

reaching a given state starting at 8,0,0 is the number printed after a dot following the trio of numbers labeled PARENT in the line corresponding to the given state. This is true for all the states. Thus, for example, to measure 2 gallons, the first state that has a 2 as a component is 3,2,3 whose distance is given in the printed output as 2. Thus, it is possible to measure 2 gallons in two pourings: 1) pour 5 gallons into the 5-gallon jug from the 8-gallon jug and 2) pour 3 gallons into the 3-gallon jug from the 5-gallon jug, leaving 2 gallons in the 5-gallon jug. The number of pourings required for reaching each state is shown opposite the corresponding oval in Fig. 3. The way to determine the sequence of pourings without constructing the tree of Fig. 3 is locating the final state in the column of PARENTS that first fulfills the condition desired. Then trace the sequence of pourings in reverse order as follows: searching above the line where the PARENT was located, look for the same state as one of the children (trios appearing in each line after two dots.) It will only appear once. The PARENT of this state is at the beginning of the line. Look above among the children states for this new PARENT and locate its parent and continue with the process until the initial state is reached. The sequence of PARENTS gives the sequence of pourings in reverse order.

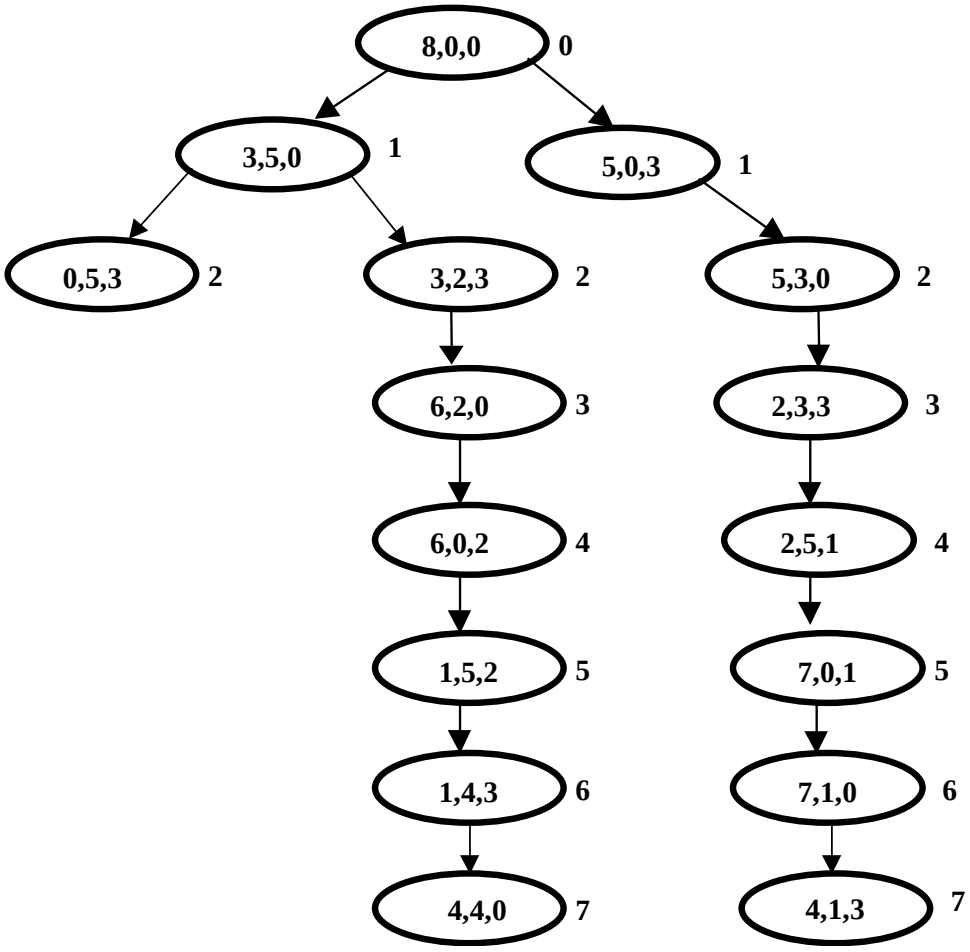

Figure 3.

Although puzzles with small capacities of the jugs can be solved by hand either by trial and error or using the techniques presented, when the puzzle involves larger capacities the computer program exhibits its real power. For instance, the solution of a puzzle calling for measuring 35 gallons with jugs with capacities 69, 37 and 32 would be very laborious to solve by trial and error, or by building the state graph. The program, however, gives in a few seconds the following sequence of states:

1 PARENT= 69 0 0 . 0 ..32 37 0 .37 0 32 .
2 PARENT= 32 37 0 . 1 ..0 37 32 .32 5 32 .

3 PARENT= 37 0 32 . 1 ..37 32 0 .
4 PARENT= 0 37 32 . 2 ..
5 PARENT= 32 5 32 . 2 ..64 5 0 .
6 PARENT= 37 32 0 . 2 ..5 32 32 .
7 PARENT= 64 5 0 . 3 ..64 0 5 .
8 PARENT= 5 32 32 . 3 ..5 37 27 .
9 PARENT= 64 0 5 . 4 ..27 37 5 .
10 PARENT= 5 37 27 . 4 ..42 0 27 .
11 PARENT= 27 37 5 . 5 ..27 10 32 .
12 PARENT= 42 0 27 . 5 ..42 27 0 .
13 PARENT= 27 10 32 . 6 ..59 10 0 .
14 PARENT= 42 27 0 . 6 ..10 27 32 .
15 PARENT= 59 10 0 . 7 ..59 0 10 .
16 PARENT= 10 27 32 . 7 ..10 37 22 .
17 PARENT= 59 0 10 . 8 ..22 37 10 .
18 PARENT= 10 37 22 . 8 ..47 0 22 .
19 PARENT= 22 37 10 . 9 ..22 15 32 .
20 PARENT= 47 0 22 . 9 ..47 22 0 .
21 PARENT= 22 15 32 . 10 ..54 15 0 .
22 PARENT= 47 22 0 . 10 ..15 22 32 .
23 PARENT= 54 15 0 . 11 ..54 0 15 .
24 PARENT= 15 22 32 . 11 ..15 37 17 .
25 PARENT= 54 0 15 . 12 ..17 37 15 .
26 PARENT= 15 37 17 . 12 ..52 0 17 .
27 PARENT= 17 37 15 . 13 ..17 20 32 .
28 PARENT= 52 0 17 . 13 ..52 17 0 .
29 PARENT= 17 20 32 . 14 ..49 20 0 .
30 PARENT= 52 17 0 . 14 ..20 17 32 .
31 PARENT= 49 20 0 . 15 ..49 0 20 .
32 PARENT= 20 17 32 . 15 ..20 37 12 .
33 PARENT= 49 0 20 . 16 ..12 37 20 .
34 PARENT= 20 37 12 . 16 ..57 0 12 .
35 PARENT= 12 37 20 . 17 ..12 25 32 .
36 PARENT= 57 0 12 . 17 ..57 12 0 .
37 PARENT= 12 25 32 . 18 ..44 25 0 .
38 PARENT= 57 12 0 . 18 ..25 12 32 .
39 PARENT= 44 25 0 . 19 ..44 0 25 .
40 PARENT= 25 12 32 . 19 ..25 37 7 .
41 PARENT= 44 0 25 . 20 ..7 37 25 .
42 PARENT= 25 37 7 . 20 ..62 0 7 .
43 PARENT= 7 37 25 . 21 ..7 30 32 .
44 PARENT= 62 0 7 . 21 ..62 7 0 .
45 PARENT= 7 30 32 . 22 ..39 30 0 .
46 PARENT= 62 7 0 . 22 ..30 7 32 .
47 PARENT= 39 30 0 . 23 ..39 0 30 .
48 PARENT= 30 7 32 . 23 ..30 37 2 .
49 PARENT= 39 0 30 . 24 ..2 37 30 .
50 PARENT= 30 37 2 . 24 ..67 0 2 .
51 PARENT= 2 37 30 . 25 ..2 35 32 .
52 PARENT= 67 0 2 . 25 ..67 2 0 .
53 PARENT= 2 35 32 . 26 ..34 35 0 .
54 PARENT= 67 2 0 . 26 ..35 2 32 .

55 PARENT= 34 35 0 . 27 ..34 3 32 .
56 PARENT= 35 2 32 . 27 ..35 34 0 .
57 PARENT= 34 3 32 . 28 ..66 3 0 .
58 PARENT= 35 34 0 . 28 ..3 34 32 .

To save space we show only 58 out of 138 parent states produced by the program. State # 51 is the first having a component 35. By tracing back the PARENTS as explained above, we deduce that a solution to the puzzle in a minimum number of pourings is given by the following sequence of 27 states (26 pourings):

(69,0,0), (32,37,0), (32,5,32), (64,5,0), (64,0,5), (27,37,5), (27,10,32), (59,10,0), (59,0,10), (22,37,10), (22,15,32), (54,15,0), (54,0,15), (17,37,15), (17,20,32), (49,20,0), (49,0,20), (12,37,20), (12,25,32), (44,25,0), (44,0,25), (7,37,25), (7,30,32), (39,30,0), (39,0,30), (2,37,30), (2,35,32).

8. MINIMIZING POURINGS AND OTHER CRITERIA

Can we solve the puzzle in less than 7 pourings? It turns out, as we shall see presently, that we cannot, because the program STATES implements a solution to a particular case called Separation Between Two Vertices of the more general problem called Shortest Route Problem. The Separation Problem coincides with the Shortest Route Problem for the case in which each edge has unit length [7].

To discuss the question of what is the minimum number of pourings to solve the puzzle we will introduce the Shortest Route Problem and relate it to the pouring problem. In Fig. 2 we could associate with each edge of the graph one or more numbers. There are several possible choices. For example, we could associate a 1 to indicate that each edge represents one pouring . We could also associate a number between 1 and 8 depending on the amount of wine in gallons of the jug that is lifted to do the pouring. Now comes a powerful idea. We could reinterpret the state diagram as a schematical road map that connects cities and the numbers associated with the edges as though they were lengths of the roads in miles. If we associate the number 1 with each edge, we could pose the problem: What is the shortest possible route we could follow from “city” (8,0,0) to “city” (4,4,0)? We are allowed only to traverse the roads (edges) in the directions of the arrows. This problem corresponds to the puzzle: What pourings should be made to divide the 8 gallons into two halves so that the number of pourings is minimum. If on the other hand we associated to each edge the amount of wine in the jug from which we pour the wine, then the shortest route problem is associated with the following variation of the puzzle: What is the sequence of pourings that should be made to divide the 8 original gallons in two halves in such a manner that the person doing the pourings minimize the total effort of lifting the jugs, with effort being measured as the sum of the weights lifted in the pourings. (Here we assume the weights of the jugs are negligible, otherwise these weights should be added to the weight of the wine for each pouring and assigned to the corresponding edges in the graph.)

Distance is not the only interpretation we can give to the numbers associated with the edges of a graph. Another possible interpretation is connected with the flow of materials; the numbers could represent cost per unit of flow. The flow in different settings could be flow of water, oil, electricity, money, information (bits), refrigerators, cement, cars, music, documents, etc. A third interpretation of the numbers associated with the edges could be capacity of flow. It is also possible to associate more than one number to each edge, for instance, the first could be the time of traverse and the second capacity.

9. SHORTES ROUTE ALGORITHM

There are many algorithms to solve the shortest route problem in an oriented graph [9]. For the particular case of the Separation Problem between an Origin Vertex and a Destination Vertex the following algorithm can be used [7]:

Every vertex X_i is progressively given a number m starting by giving the Origin Vertex the number 0; all vertices that can be reached from the origin by traversing (in the correct direction indicated by the arrow in the edge) a single edge are given the number 1; all the vertices that have not already been labeled and that can be reached by

traversing in a single edge starting from a vertex labelled 1 are given the number 2. The process is continued increasing by one the number given to the nodes at each stage until the destination vertex is labelled. The label received by the destination vertex is the minimum distance between the origin vertex and the destination vertex. The shortest route can be found in reverse order by starting at the destination vertex and going through vertices with numbers decreasing by one. There is always a vertex with a number diminished by one next to (going in the direction opposite to the arrow) any vertex since the node received its labelling from one such node. This method was illustrated when travelling from children to PARENTS in discussing the Table of PARENTS. An example of a graph where this algorithm has been applied is shown in Fig. 4.

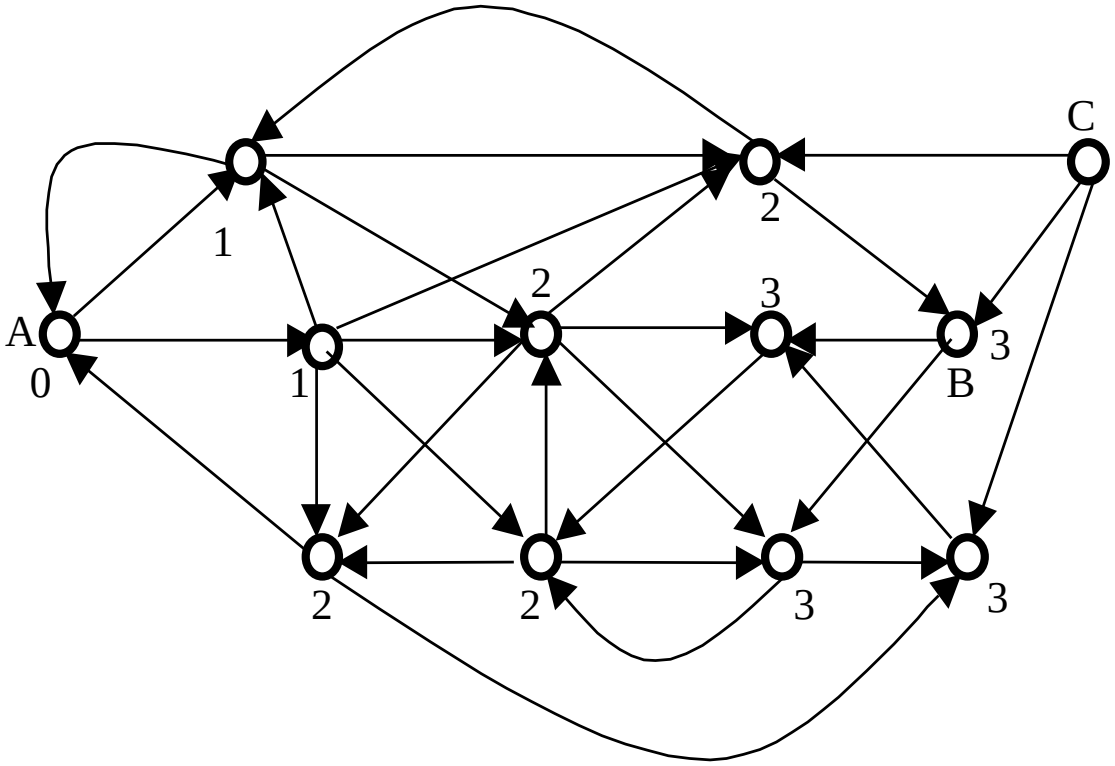

Figure 4.

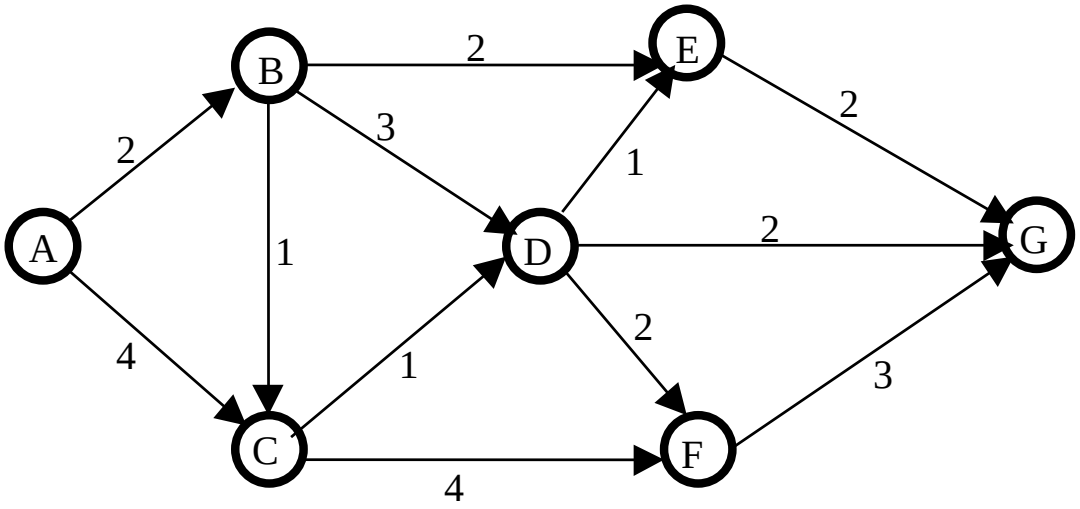

Figure 5a.

When all labels that are possible to assign have been assigned, if there remains any node unlabelled, such a node cannot be reached from the starting node by traversing the edges in the directions of the arrows. Such is the case of node C in Fig. 4. Note that if the destination node is changed, the labeling process and the numbers given to the vertices do not change, although they could stop sooner or take longer. This means that if the algorithm is applied until no new nodes can be labeled, one can find the shortest routes between the origin vertex and all other vertices in the graph.

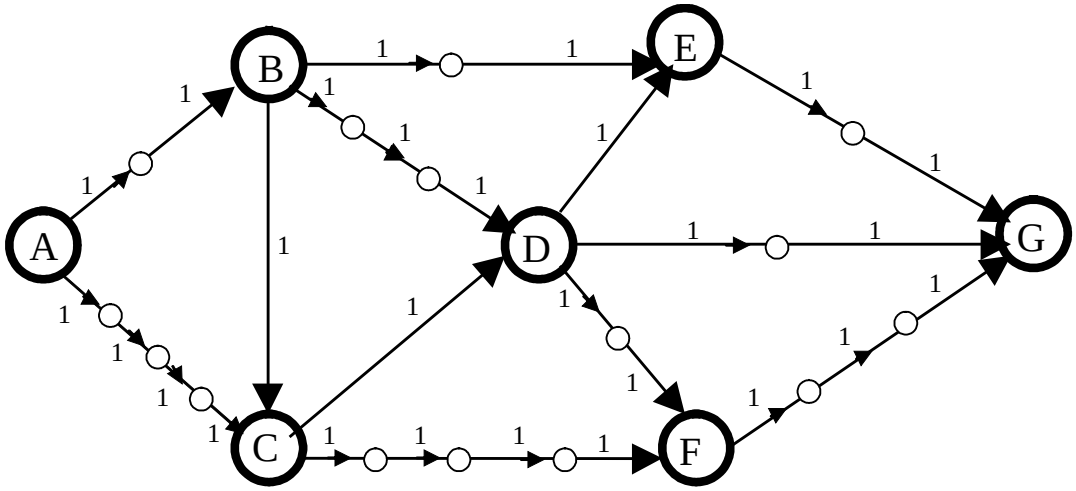

Figure 5b.

It is quite simple to prove that the algorithm given is correct, that is, it does indeed determine the shortest route between the origin vertex and all vertices that are given labels. The proof is by contradiction. Suppose the algorithm was properly applied and it ended by giving the destination vertex a number N, implying the shortest route is N units long, but that there exists a route starting from the origin node which can be traversed in the direction of the arrows of the edges which has length M < N (that is, there is a shorter route.) Since all edges of the graph have unit length, the route must go through M + 1 nodes which we will call 0' (origin), 1', 2', ..., M'. Now, since node 1' can be reached from the origin through a single edge we should have labeled node 1' with a 1; similarly since 2' can be reached from node 1' which has label 1 with one edge, it should be labeled with a 2. Continuing in the same manner M' should have received a label M. But since M < N, then the algorithm was not properly applied. The contradiction shows there can not be any route which is shorter than the one determined by the algorithm.

Let us now move to the case in which the edges of the graph are not necessarily of unit length. We assume that the lengths of the edges are positive integers as in Fig. 5a. Through the device of introducing artificial nodes we could transform the graph into one having all the edges of unit length as is illustrated in Fig. 5b.

If the lengths of the edges are not integers but positive real numbers we could approximate them to computer accuracy by using positive floating point numbers. Now by taking the unit distance sufficiently small we could express all lengths with integers and again transform the graph by introducing artificial vertices into a graph in which all edges have unit length (in terms of the unit distance mentioned.) The reader needs not to worry, all this is mentioned only for the purpose of theoretical analysis, since it would generally become unwieldy to apply the separation algorithm to any but the simplest graphs having small integers as lengths. We can however through a thought experiment deduce the practical algorithm that would be applied to a real graph. Let us explore with the simple graph of Fig. 5 a.

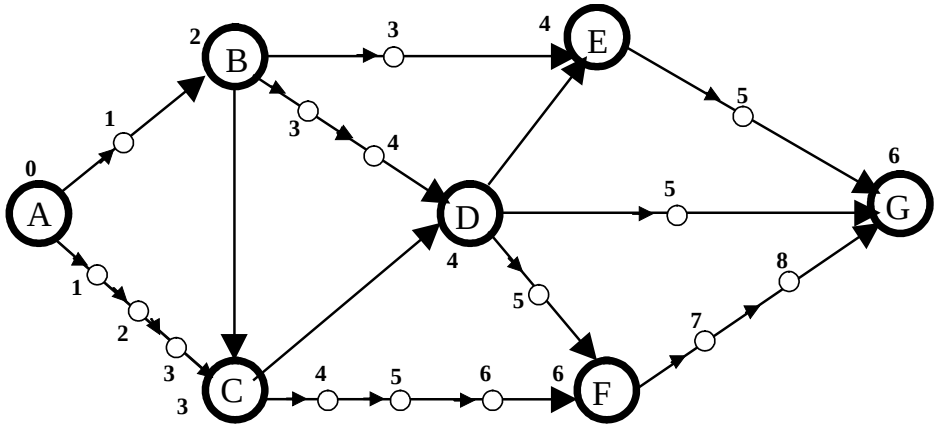

Figure 6.

In Fig. 6 we repeat Fig. 5b having labeled all the nodes (real and virtual) according to the separation algorithm. Node C was given a label 3 because by the time node 3 in edge A.1.2.3.C tries to label C with a 4, it is already labeled 3 via vertex B. Node D is labeled 4 because by the time node 4 in the edge B.3.4.D tried to label it with a 5 it is already labeled 4 via C. If we observe the labeling process carefully, we arrive to the conclusion that one is advancing in "concentric circles, " that is vertices at equal distances from the origin node. Thus we can make the following simplification: The virtual vertices can be skipped and the labeling of the real vertices done directly by adding to the label of a node whose neighbors are being labeled the length of the connecting edge having an arrow in the proper direction. However, in contrast to the separation algorithm, a label once given to a node does not necessarily remain, because maybe the sum of two sides such as AB and BC are less than the other side AC (this cannot happen in geometric triangles in ordinary euclidean space but it can happen in schematic graphs where the position of the nodes may be arbitrary, not necessarily associated with space, and where numbers on edges may represent time of traverse, cost, capacity, etc.) Whenever the sum of the lengths of two sides of a triangle are less than the length of the third side, the label in a node will change to the sum of the two. It may change again if another chain of edges has a sum of lengths smaller than the length of the direct connection from a labeling node previously encountered.

When are we sure that the label of a node will no longer change? If we analyze the situation carefully, we will note that the node P with the smallest label at the beginning of each labeling stage will not change again, because any new possible label for a node will be at least as large as the label of P plus the length of the shortest edge in the graph. If all the lengths are positive then the smallest label cannot be improved with subsequent labelings and we can stop considering such a node in the process as soon as it labels its neighbors.

We have deduced the Shortest Route Algorithm (that avoids altogether virtual nodes) first discovered by E. Dijkstra [8] that goes as follows:

1. Give the origin node the label 0 and mark it pivotal. To all other nodes give them the label ∞ (infinity) and mark them temporal .
2. Label all the direct neighbors of the pivotal node (those that can be reached through an edge traversed in the proper direction) that are temporal with the lesser of the following two numbers: the old label of the node being labeled, or the sum of the label of the pivotal node plus the length of the edge connecting the pivotal node with the node being labeled. If the second number is the lesser add a second part to each label with the name of the pivotal node.
3. Mark the label of the pivotal node permanent and erase its pivotal character.
4. Find among the nodes with temporal labels the one with the smallest label. Mark the corresponding node pivotal. (In case of ties any one of them will do.) If the pivotal node is the destination node, go to step 5, otherwise return to step 2.

Stop. The label of the destination node is the length of the shortest route. The route can be traversed in reverse by going to the nodes indicated by the second parts of the labels starting with the destination node until the origin node is reached.

NOTES: The length of the shortest route from the origin node to each one of the nodes that have permanent labels is given by the corresponding label. The routes can be found in reverse order via the second parts of the labels (as was done for the destination node.) The nodes with temporal labels are at a distance from the origin node equal or larger than the destination node but their labels do not necessarily have the correct distance until the labels become permanent. The process could be continued until all temporal labels become permanent and then all labels indicate the shortest distance of the origin node to the node in question and the routes can be traversed in reverse following the second parts of the labels. Any node that does not evenually receive a label other than the initial ∞, is not accesible from the origin node. (It is either disconnected or else connected via a set of edges with arrows in directions that impede reaching the node.)

10. AUTOMATING DIJKSTRA´S ALGORITHM

To automate Dijkstra's Algorithm in QBASIC we start by designing a data structure that describes the graph for which the shortest route is desired. To take advantage of the infrastructure associated with strings in QBASIC, which have flexible length up to 256 characters, we will assume that we will work with graphs having at most 256 nodes and that the edges of the graphs have integer lengths ranging from 0 to 255. We will use a linear string array **g$** called the array of succesors in which each element of the array corresponds to a node. The nodes will be numbered from 1 (origin) to **nn** (destination.) In each element of the array we will store characters whose ASCII codes are the numbers corresponding to the succesors (direct neighbors) of the node corresponding to the element. In a second string array **ll$,** called the array of lengths, in corresponding places we will store the lengths of the edges connecting each node to their succesors. Thus, for the graph of Fig. 5a, the string arrays **g$** and **ll$** are:

g$	ll$
1|2 3	1|2 4
2|3 4 5	2|1 3 2
3|4 6	3|1 4
4|5 6 7	4|1 2 2
5|7	5|2
6|7	6|3
7|	7|

where the numbers to the left of the vertical lines are the indexes of the elements (not stored) and the numbers to the right are the ascii codes of the contents of the elements. The nodes of Fig. 5a have been numbered in alphabetical order, that is A = 1, B = 2, C = 3, D = 4, E = 5, F = 6, G = 7.

From the second element of **g$** we see that the direct neighbors of node 2 are 3, 4, and 5. From the corresponding elements of the array **ll$** we see that the lengths of the edges that go from node 2 to nodes 3, 4 and 5 are respectively 1, 3 and 2. The reader should check that all the information of Fig. 5a is stored in the string arrays **g$** and **ll$.**

To distinguish permanent labels from temporary labels we will use a numerical array **p** with **nn** elements; those elements having a 0 correspond to the temporary labels. Permanent labels will have a –1 in the corresponding element of array **p.** The labels of Dijkstra's algorithm will be stored in a numerical array **lab.** Infinity will be represented by the floating point number 1E+38. The second parts of the labels pointing to predecessors in the shortest routes will be stored in a numerical array **ap.** In the numerical variable **np** the pivotal node will be stored . The description of the information corresponding to a graph will be put in DATA statements. In the first statement the number of nodes of the graph (7 for the graph of Fig. 5a) is written. For each of the nodes in sequential order a DATA line with the following information is written:

number of successors, successor, length, successor, length, ..., successor, length

The number of pairs : *successor, length* coincides with the number of successors at the beginning of each line. If a node has no successors only the number 0 appears in the line. The numbers of successors for each node are stored in the elements of a linear array **nnss.**

The automation of Dijstra's algorithm is achieved by the following commented QBASIC program:

```
REM Number nodes so that 1 is origin and nn is destiny
READ nn: REM nn=number of nodes in graph
DIM g$(nn), ll$(nn), lab(nn), nnss(nn), p(nn), ap(nn)
REM g$=array of successors, ll$=array of lengths, lab=array of labels,
REM p=array of permanence of label, ap=array of pointers to predecessors

FOR i = 1 TO nn
p(i) = 0: lab(i) = 1E+38: REM all nodes start with infinite temporary labels
ap(i) = 0: READ ns: nnss(i) = ns: REM ns=number of successors
lab(i) = 1E+38

FOR j = 1 TO ns
IF ns > 0 THEN READ s, l: g$(i) = g$(i) + CHR$(s): ll$(i) = ll$(i) + CHR$(l)
REM s=successor, l=length
NEXT j
NEXT i

np = 1: REM origin node made pivotal and its label set to 0. np=pivotal node

lab(np) = 0: p(np) = -1
DO
FOR i = 1 TO nnss(np)
x = lab(np) + ASC(MID$(ll$(np), i, 1)): ind = ASC(MID$(g$(np), i, 1))
IF p(ind) >= 0 AND lab(ind) > x THEN lab(ind) = x: ap(ind) = np
NEXT i

REM obtains node jnd with minimum lab(jnd)
mn = 1E+38: jnd = 0
FOR j = 1 TO nn
IF p(j) >= 0 THEN IF mn > lab(j) THEN mn = lab(j): jnd = j
NEXT j
np = jnd
p(jnd) = -1
LOOP UNTIL jnd = nn

REM prints nnss array

LPRINT : LPRINT "Numbers Of Successors"
FOR i = 1 TO nn
LPRINT i, nnss(i)
NEXT i

REM prints g$ array
LPRINT : LPRINT "Successors$"
```

```
FOR i = 1 TO nn
LPRINT i,
FOR j = 1 TO LEN(g$(i))
LPRINT ASC(MID$(g$(i), j, 1));
NEXT j
LPRINT
NEXT i

REM prints ll$ array
LPRINT : LPRINT "Lengths$"
FOR i = 1 TO nn
LPRINT i,
FOR j = 1 TO LEN(g$(i))
LPRINT ASC(MID$(ll$(i), j, 1));
NEXT j
LPRINT
NEXT i

REM prints lab array containing labels
LPRINT : LPRINT "Labels"
FOR i = 1 TO nn
LPRINT i, lab(i)
NEXT i

REM prints array of pointers to predecessors ap
LPRINT : LPRINT "PredecessorPointers"
FOR i = 1 TO nn
LPRINT i, ap(i)
NEXT i
LPRINT "----------------"

REM data corresponding to example given in article (Fig. 5a)
DATA 7
DATA 2,2,2,3,4
DATA 3,3,1,4,3,5,2
DATA 2,4,1,6,4
DATA 3,5,1,6,2,7,2
DATA 1,7,2
DATA 1,7,3
DATA 0

END
```

11. LISTING 2 AUTOMATION OF DIJKSTRA´S ALGORITHM

In Listing 2 the program starts by reading the number of nodes of the graph and using that number to dimension the arrays **g$, ll$, lab, nnss, p,** and **ap.** Next the program makes all labels temporary and equal to infinity, points all predecessors to 0 (that is all nodes start with no predecessor) and reads in the DATA of the graph and constructs the array **nnss** and the string arrays **g$** and **ll$.** Once this is done, node 1 (origin) is made pivotal and its label is set to 0 and marked permanent.

Now the following operations are repeated until the destination node is marked permanent:

All the successors of the pivotal node that have temporary labels are checked to see if their labels are larger than the sum of the label of the pivotal node plus the length of the edge from the pivotal node to the node in question. If that is the case, the label is changed to the mentioned sum and **ap** is made to point to the pivotal node. From among the nodes with temporal labels the node **jnd** with the smallest label is made pivotal and its label marked permanent.

The algorithm terminates successfully when the destination node is made pivotal. The program then prints the arrays: **nnss, g$, ll$, lab,** and **ap.** At the end it prints a line of dashes to separate the printing from the next problem.

In Listing 2 the DATA lines at the end corresponds to the graph of Fig. 5a.
The functions MID$, ASC and LEN are used to manipulate and code and decode the string arrays.
A run of the program with the DATA in Listing 1 is shown below:

```
NumberOfSuccessors     (nnss)
1                 2
2                 3
3                 2
4                 3
5                 1
6                 1
7                 0


Succesors$        (g$)
1                 2  3
2                 3  4  5
3                 4  6
4                 5  6  7
5                 7
6                 7
7


Lengths$          (ll$)
1                 2  4
2                 1  3  2
3                 1  4
4                 1  2  2
5                 2
6                 3
7


Labels    (lab)
1                 0
2                 2
3                 2
4                 4
5                 4
6                 6
7                 6
```

	PredecessorPointers (**ap**)
1	0
2	1
3	2
4	3
5	2
6	4
7	4

The two arrays (**lab** and **ap**) gives us the information about the shortest routes from node 1. For example **lab**(7) = 6 means that the shortest route from node 1 to node 7 has length 6; **ap**(7) = 4 means that the shortest route from node 1 to 7 passes through node 4. Next we consult **ap**(4) = 3 which says that the shortest route from 1 to 4 before arriving to 4 passes through 3. We now consult **ap**(3) = 2 which says that the shortest route from node 1 to 3 passes through node 2 before getting to 3. We consult **ap**(2) = 1 which says that the shortest route from node 1 to 2 passes through 1 before arriving to 2. Since 1 is the origin we have finished and we can state that the shortest route from 1 to 7 passes through the nodes in order: 1; to 2; to 3; to 4; to 7. Checking the length of the shortest route we see from Fig. 5a that the lengths of the edges are as follows: (1,2) = 2, (2,3) = 1, (3,4) = 1, (4,7) = 2. Therefore the lengths of the edges in the shortest route do in fact add to 6.

With the program provided it is a profitable exercise for the reader which will familiarize him with the program and its use to execute the given program automating Dijkstra's algorithm to find the shortest route between node (8,0,0) and node (4,4,0) of Fig. 2 inputting edges weighted with the amount of gallons that have to be lifted for each pouring associated with an edge.

12. CONCLUSION

Our thesis is that for problem-solving there is no magic, general method that will solve all problems. Rather, it is necessary to teach a variety of methods that will work in different instances. It is important that the student learns to recognize families of problems that will yield to each family of solution methods. The role of the experienced teacher is to select good, broad paradigms that will cover the most important parts of the terrain that the students will be expected to move in during their future activities. The teacher should help the students to familiarize themselves with them. Some paradigms are already well established in several areas. One example is the second-order dynamic system [16]

A good source of broad paradigms are mathematical puzzles. They have the advantage of being easily remembered and requiring little specific background. One such paradigm has been developed in some detail in this article. Such important central concepts as "state", "directed graph", "shortest route", "recursion" appeared naturally in the process of solving the puzzle. Generalization of the problem and the desire to use the computer as an aid led to an excellent algorithm for the shortest route problem and the representation of a directed graph in a computer.

Some examples of puzzles which can be solved by the methods and concepts introduced in this article are "crossing problems" such as the Cannibals and Missionaries Problem [6] in which 3 cannibals and 3 missionaries traveling together have to cross a river; there is available a row boat capable of transporting two individuals (all participants know how to row.) The problem is to devise a sequence of crossings so that the cannibals never outnumber the missionaries at any moment, otherwise the life or lives of the outnumbered missionaries are put in danger of attack. Another famous puzzle that yields to state, network and shortest routes is the "12 coin problem." In this puzzle there are 12 identical-looking coins one of which is false and weighs either more or less than the other 11 good ones. The object is to use an equal arms balance to determine in 3 experiments which is the false coin and whether it is lighter or heavier than the good ones. Of a similar nature is the problem of achieving a given configuration of numbers in the game of "fifteen," which consists of a square containing 15 numbered sliding blocks and an empty space (to provide the opportunity to slide the blocks near it) arranged in a 4×4 array. Through sliding operations the purpose of the game is to go from a standard initial configuration (for example the blocks numbered consecutively from 1 to 15 by rows) to a given final configuration (for example the blocks numbered consecutively by columns.) The state can be

defined as a vector with 16 numbers (the empty space can be given the number 16) giving the position of each block, say, by rows, and each transition from one state to another corresponds to displacing the empty space in one of four directions: up, down, left, right, with some directions forbidden for certain states in the edges of the square. By writing a program similar to the one given in the article, the puzzle can be solved automatically in a minimum number of sliding operations. Artificial Intelligence researchers are interested in getting to the goal state without having to expand all the new states that are generated by introducing *heuristics.* This leads to the analysis of such strategies as *breath first, depth first* and *best first.* [10]. Designing optimum medical diagnostic trees as well as optimum diagnostic trees for communication systems, computer systems and others is also a problem that can be modelled with states, networks and shortest-routes [11,14]. The same concepts can be applied in the design of optimum codes for communications, [13], some search strategies in data banks and processing strategies for merging records of computer files [15].

Network problems in which finding the shortest route is useful are bountiful [12]. There are, of course, the problems of transportation networks in which a vehicle operator wants to know which is the less expensive route to go from one city to another. Industrial processes are often modeled using networks; routes between different nodes may represent various courses of action, each edge corresponding to buying raw materials, storing them, processing them in factories, storing finished products, transporting them to various cities, storing and finally selling them. Each edge has associated a cost or revenue related to the transition between different states of the process. The shortest route represents the most economically convenient course of action.

Finding the shortest route in a network is often a subproblem of more general problems such as finding the most economic way of shipping merchandise between sources and destinies in networks with maximum capacities of flow in the edges. The different edges have different unit costs of shipping.

It is said that the success of Bobby Fisher as a former World Champion in Chess had to do with the fact that he forced his opponents to play the game in almost unexplored areas of the game where he excelled because he was a very original chess problem solver. Most other champions stick to well explored terrains where good strategies have been thoroughly analyzed by international masters, many of them being so well known that they have names such as the "two knight defense", the "indian defense", and so forth. If we were able to collect a set of paradigms that properly cover the field of problem-solving in the field of mathematics, we would stand a better chance of producing Bobby Fishers in mathematics. The author feels that papers exhibiting and discussing such paradigms in the detail provided in this paper would be a welcome addition to the literature of the art of teaching problem-solving.

13. REFERENCES

[1] Euler L., "The Seven Bridges of Königsberg ," in J. R. Newman (Ed.) *Men and Numbers, Vol. I of the World of Mathematics,* Simon and Schuster, New York, 1956, pp. 573 – 580.

[2] Turnbull H. W., "The Great Mathematicians," in J. R. Newman (Ed.), *Men and Numbers, Vol. I of The World of Mathematics,* Simon & Schuster, New York, 1956, pp. 75 – 178.

[3] Ore O., "Pascal and the Invention of Probability Theory," *Americal Mathematical Monthly, Vol. 67,* 1960, pp. 409 – 416.

[4] Newell A., Shaw J. C. & Simon H. A., "Chess Playing Programs and the Problem of Complexity," in E. A. Feigenbaum & Feldman J. *Computers and Thought,* Mc Graw-Hill Book Company, New York, 1963, pp. 39 – 70.

[5] Kaufmann A. & Faure R., *Introduction to Operations Research,* Academic Press, New York, 1968.

[6] Bellman R., Cooke K. L. & Lockett J. A., *Algorithms, Graphs and Computers,* Academic Press, New York, 1970.

[7] Kaufmann A., *Graphs, Dynamic Programming and Finite Games,* Academic Press, New York, 1967, pp. 255

[8] Dijkstra E. W., "A Note on Two Problems in Connection with Graphs," *Numerische Matematik, Vol. 1,* 1959, pp. 269 – 271.

[9] Dreyfus S., *An Appraisal of Some Shortest Path Algorithms, RM-5433-PR,* Rand Corporation, Santa Monica, CA, Oct., 1967.

[10] Nilsson N. J., *Problem-Solving Methods in Artificial Intelligence,* Mc Graw-Hill Book Company, New York, 1971.

[11] Murray-Lasso M. A. & Lara J., "Diseño de Árboles Óptimos de Diagnóstico con Métodos de Inteligencia Artificial," *Memoria del Primer Congreso de la Academia Nacional de Ingeniería,* Guanajuato, 23 - 24 de junio, 1975, pp. 179 – 193.

[12] Ahuja R. K., Magnanti T. L. & Orlin J. B., *Network Flows: Theory, Algorithms and Applications,* Prentice-Hall, Upper Saddle River, NJ., 1993.

[13] Huffman D. A., "A Method for the Construction of Minimum Redundancy Codes," *Proceedings I. R. E., Vol. 40,* no. 10, September, 1952, pp. 1098 – 1101.

[14] Murray-Lasso M. A., "Aplicación de la Metodología de Árboles de Decisión e Información al Diagnóstico Médico," *Revista Mexicana de Electricidad, Vol. 32,* No. 383, 1972, pp.- 405 – 407, agosto, y No. 384, pp. 447 – 451 y 478 – 479, septiembre. Also appeared in *Boletín del Instituto Mexicano de Planeación y Operación de Sistemas, Vol. 3,* No. 14, nov. – dic. pp. 12 – 32.

[15] Knuth D. E., *The Art of Computer Programming. Vol. III, Sorting and Searching,* Addison-WesleyPublishing Company, Reading, MA, 1973, pp. 365.

[16] Cannon R. H. Jr., *Dynamics of Physical Systems,* Mc Graw-Hill Book Company, New York, 1967.

Author Byography

Marco A. Murray- Lasso

Born in Mexico City, Mexico September 1, 1937. He is a Mechanical - Electrical Engineer, in 1960, by the UNAM (National Autonomous University of Mexico), Mexico City. Murray Lasso obtained his M.Sc. degree in Electrical Engineering in 1962 from Massachusetts Institute of Technology Cambridge, MA. He got his Ph. D. in Automatic Control, in 1963, at the Massachusetts Institute of Technology, Cambridge, MA.

He is currently professor of Systems Engineering, Honor Member and Founder President of the National Academy of Engineering, Founding member of the Mexican Academy of Technology, Member of the Mexican Academy of Informatics, Founding member of the Mexican Academy of Sciences, Arts, Technology and Humanities, Former President of the Council of Academies of Engineering Technological Sciences (CAETS)