

THE LOGIC OF COMPUTER LOGIC PUZZLES

John Higgins

School of Education, University of Bristol

1991

ABSTRACT: This paper describes the structure of a set of computer logic puzzle generators. It discusses the building of 'intelligent' solving routines in which the machine tries to draw inferences, using only the information that it has so far displayed so that it can announce when a solution is possible; one of the debugging problems that arose during the development stage is described.

BACKGROUND

The logic puzzles described below were developed as one element in English courses for overseas students. The justification for using puzzles in language teaching is that their solutions are often numeric or spatial, but the clues are presented as text; the learners' task therefore is to translate text into some non-linguistic representation so that inferences can be drawn. In the process learners are exposed to text but are concentrating on meaning rather than on the linguistic forms. Moreover, if the task is turned into a group activity, a great deal of spoken language occurs incidentally, especially if the participants each receive only some of the clues and have to ask questions or read aloud their own clues in order to share information. This is very much in tune with current language learning pedagogy, which emphasises the role of meaning in the learning process and the value of creating an 'information gap' so that communication is necessary for some reason other than the teacher's say-so.

The puzzles themselves are not usually of great difficulty; after all they were designed for people with only an intermediate command of English. Each puzzle tends to have a linguistic focus so that a teacher can tie them in to a syllabus or draw attention to some point of usage when the activity is over. In this sense TRACK, for example, is about prepositions; CARFAX is about modal verbs; TRIP is about past tenses, and INVENTION is about comparisons. (See appendix for a full list and description of the puzzles and sample runs.)

FORMS OF OUTPUT

Using a computer to create the puzzles has the great advantage that a number of near replicas of a basic puzzle form can be created with almost no effort beyond writing the initial program. All the programs except SEESAW were written so that the output may be a paper puzzle sent straight to the printer or an interactive puzzle to be solved piecemeal by calling up clues on screen, keying in partial answers and getting feedback.

Apart from the central logic, each program consists of a set of display routines which print out an attractive screen or page to give the context of the puzzle, and the necessary interactive routines to get and interpret the user's key presses. The programs each draw on a stored set of scenarios, i.e. sets of personal or place names, so that consecutive runs are different. In programming terms drawing the screens, designing menus and input routines, and choosing the scenarios are all relatively trivial tasks, but it is well worth taking care over them in order to make the user's task as easy as possible and to ensure that good design principles are followed. These elements probably account for well over half the time and effort that go into the creation of the programs.

PROGRAM CONTENT

The meat of each program is a matrix for the problem, stored as an array of numbers each of which points to an entry in a string array of names or words. A random number generator is used to write a solution into the array but the solution is not, of course, displayed. The use of randomisers both here and at the cluing stage probably means that each program will create several million puzzles before repeating itself.

The program then calls up one of several clue generators which display a true sentence, using information from the matrix. We tend to put between five and ten of these clue generators into a program, but there may be random elements within the generator so that more than a dozen different sentence types are produced.

In TRACK, the program which asks you to identify station names on a railway map, the eight clue generators are as follows:

NORTHORSOUTH

(PEARL) is due/further north/south of/than (CORAL).

EASTORWEST

(PEARL) is due/further east/west of/than (TOPAZ).

BETWEEN

(OPAL) is between (PEARL) and (TOPAZ).

NEXTTO

(PEARL) is next to one/two/three other stations.

TERMINUS

(PEARL) is/is not a terminus.

TWOSTATIONS

(PEARL) is two stations away from (TOPAZ).

BRANCH

(PEARL) is/is not on (either) the northern/southern/ northern or southern branch.

FURTHEST

(PEARL) is/is not (one of) the furthest east/west/east or west stations.

There is only one form of the map at the moment, though the authors are working on a new version which will be able to display several different maps. In the current version stations are referenced by the program with the following numbers:

1	2	3		
			4	5
8	7	6		

GENERATING CLUES

To create a clue the program takes the next number off the top of a jumbled list of numbers. Let us say it finds number 1. It looks this up in the table of names and discovers the station name is PEARL. It now randomly selects a clue type, say NORTHORSOUTH. In that sub-program it randomly chooses between due and further, let us say choosing due on this occasion. Since PEARL is already on the northern branch, it is forced to choose north. The next line of the program tells it that with combination of Station 1 and due, the choice of Station 8 is forced, so it looks up Station 8 and supplies the complete sentence “PEARL

is due north of CORAL.” Had the word been *further* rather than *due*, it would have randomly chosen one of the stations from 4 to 8.

To ensure that the program does not repeat clues or supply too much redundant information, there is a separate 8 by 8 flag matrix which stores a record of the clues which have been given or should not be given. Initially the BETWEEN clue is flagged out for stations 1, 5 and 8, since these stations are not between any two others. Once the *due north of* clue has been given for Station 1, both NORTHORSOUTH and BRANCH are flagged out for stations 1 and 8, since the clue has implied which branch PEARL and CORAL (in our example) are on. The NEXTTO clue for any station flags out TERMINUS and FURTHEST as well. TERMINUS flags out FURTHEST and vice versa, and both TERMINUS and FURTHEST flag out NEXTTO if the clue has the form *is a terminus* or *is one of the furthest west stations* but not if the clue is *isn't a terminus* or *is not the furthest east or the furthest west*, since in those cases the NEXTTO clue will supply extra information.

SOLUTIONS

The number of clues that will lead to a unique solution varies. The smallest number of clues that the TRACK program has ever produced spontaneously is five. Normally a solution requires between eight and twelve, and occasionally up to twenty clues. The program has a potential 64 clues to display (less any which have been flagged) before it runs out of new information. Should it reach that limit, the flags in the storage matrix are reset and the program continues displaying clues which repeat known information, but that has never happened to the program in use.

The number of clues required depends, of course, on the nature of the puzzle. CARFAX, the simplest, is normally solved within four clues, while INVENTION can be solved in five and rarely needs more than eight.

If the program does no more than select and display clues, it leaves the human solver not only to draw inferences but also to decide when enough information has been given to find a solution. The solver can enter a solution and the program can then match it against the matrix and report success or failure, but the program itself does not know if the solver was justified in making the attempt at that point; perhaps the solver had too little information and was guessing or had already been given redundant information. Some of our programs have stopped at this stage. They are puzzle generation aids rather than full puzzle creators. However, in three cases, namely INVENTION, TRACK and CARFAX, we have gone further and have written intelligent solving routines which allow the program to know when a solution is possible.

(It is, by the way, debatable in what sense a computer can ‘know’ anything, but that discussion is beyond the scope of this paper.)

THE SOLVER ROUTINE

One immediate benefit of the intelligent solving routine is that one can let the program generate paper puzzles automatically; without it a human must intervene to decide when enough clues have been printed. Another is that the program can signal when it has discovered either a partial or a complete solution, thus alerting a solver at the keyboard so that he or she can look back through the clues already given and look for inferences to be drawn from combinations of clues.

The mechanism of the solver sub-program can best be illustrated with the solver I wrote for the TRACK puzzle. This, like the main clue generator, is based on a matrix, but this time using the process known as masking or ANDing. At the start of the program an array of 8 numbers, one for each station, is declared and each number is set to 255, which is binary 11111111. Within each clue generator, the implications of each form of the clue are represented by numbers which indicate in binary notation which stations are possible locations for the name used in the clue. In the TRACK example above, the sentence “PEARL is due north of CORAL” produces the numbers 7 for the first variable (the number which output PEARL) and 224 for the second variable (the one which output CORAL). 7 is binary 00000111, representing the fact that stations 1, 2 and 3 are possible locations. Pearl’s entry in the solving array is ANDed with 7. If it is the first clue, this produces the result

	11111111
AND	00000111
IMPLIES	00000111

Similarly CORAL’s entry is ANDed with 224, whose binary form represents the fact that CORAL can only be station 6, 7 or 8.

	11111111
AND	11100000
IMPLIES	11100000

The results of these operations replace the numbers in the solving array. If later we get the clue “PEARL is a terminus”, the current value for PEARL is ANDed with 145, binary 10010001 (meaning stations 1, 5 and 8 are possible locations).

	00000111
AND	10010001
IMPLIES	00000001

We have now reached a numeric representation of the fact that there is only one possible location for PEARL, something which we can express in words as “If PEARL is a terminus and PEARL is due north of something, it must be the western terminus of the northern branch”.

THE SOLVER SUB-PROGRAM

As well as writing all the implications into the solving array, the program needs some way of deciding when to take action. This is done by scanning the solving array after each clue. The scanning must take place in two directions, left-to-right in order to find out if all locations but one have been eliminated for a station, and top-to-bottom to discover if all stations but one have been eliminated for a location. The horizontal scan is done by straightforward comparison of the numbers with successive powers of 2. The vertical scan is done by ANDing each number in turn with the relevant power of 2 to find out if this bit is set in the number. If this succeeds (i.e. returns the same power of 2) only once, then the location has been identified by elimination. In this operation the results of ANDing are not stored in the solving array.

HISTORY

If PEARL is due north of CORAL, and PEARL is later identified as station 1, then we also have new information available about CORAL. To make use of this, the program has to store a history of the clues that have been issued so that, as soon as a station is identified, the program can re-run all clues which mentioned it. The re-run is not necessary with the one-variable clues like TERMINUS and FURTHEST, but is necessary for all the two-variable and three-variable clues, namely NORTHORSOUTH, EASTORWEST, TWOSTATIONS, and BETWEEN. The history attached to each station is of unpredictable length and is therefore stored in string form in a list rather than in an array.

DEBUGGING

It was this history component which turned out to be the most troublesome part of the program to write and debug. The problem arose since the re-runs are background operations; they do not yield screen output. Therefore any anomalies tend to show up not at the time but in unpredictable events occurring much later.

When all the solver routines were complete (as I thought), I discovered that about one time in three the program was terminating too soon, i.e., promising a solution when none was possible. About one time in ten, on the other hand, it was failing to terminate at all, but would continue to issue repeated clues about just one or two of the stations. On breaking into the program when the second condition applied, I found that the value in the solving array for at least one of the stations was 0, something which I had believed to be impossible.

My first thought was that somewhere I had written a value for ANDing which was wrong and which led to a station eliminating its own true position from the array. There were well over a hundred of these numbers to be ANDed in various parts of the program. I began by checking them manually and did indeed turn up two errors. To my disappointment, correcting them did not solve the problem.

The next stage was to apply a trace. I wrote an additional routine which would copy to the printer the inputs and output of the ANDing process whenever it was carried out. A dozen runs left me with hundreds of lines of figures to work through.

The debugging process consists largely of pretending to be a computer. You abandon any notion of common sense, and turn yourself into a totally obedient slave, executing the master's commands uncomplainingly. Doing this, I eventually tracked down several points in the figures where my slavish behaviour was giving different results from the computer's. Light eventually dawned. When the program re-ran clues, it did not know which random value had been selected previously for elements like *due* and *further*. It simply reapplied the randomising process and, fifty percent of the time, was giving a different output. Correcting the error was just a matter of adding the record of these random decisions to the history and blocking the randomising process during the re-runs, making the program look in the history for a value for *due* versus *further* or whatever.

Finding the answer was at the same time both frustrating and satisfying. I had been in a sense blindingly stupid not to notice the effect of the randomiser. Probably I had conceptualised a notion of "repeat the same clue" and was beguiled by the idea of sameness into overlooking that the clues containing

randomisers would not on re-runs be “the same”. This was balanced by the eventual triumph of finding the bug and getting the program to perform reliably to specification.

Even at this stage, however, the program is still inferior to a good human solver, who can often reach a solution several clues earlier than the program reports one. The reason for this is that the program only re-runs clues when a station has been definitively identified. A human solver, on the other hand, can use pairs of clues to narrow the range of possibilities even when the information is very partial. This applies especially to the BETWEEN and TWOSTATIONS clues. If we know, for instance, that PARK is between DOCKS and FERRY and that DOCKS is on the northern branch and FERRY on the southern branch, then it is simple for us to assign PARK to station 4. The program will not get around to doing this until either DOCKS or FERRY has been placed. One could fix this by having clues re-scanned much oftener but doing so would make the program unacceptably slow.

CONCLUSION

It may be that the creator of programs of this kind gets far more fun from the act of creation than users get from using it. However, the programs have gone down well with their initial target audience, and I like to think that several classes of North African students have been introduced to an experience they might not otherwise have had and appear to have relished it.

John Higgins, December 1990

APPENDIX: The puzzles

All the following puzzles were created by John and Muriel Higgins, using a BBC B Micro and/or an IBM PC.

INVENTION: A Venn diagram of three overlapping circles representing a population. The computer prints an initial statement of the form

Out of 54 energetic students, 45 played squash, rugby and/or tennis.

and then offers clues of the form

*3 more students played only squash than played rugby and squash (but not tennis).
24 students altogether played rugby.
27 students did not play tennis.*

In the interactive form the players can fill in the number in any segment of the diagram whenever they think they know it. In the paper form a sufficiency of clues is printed out.

(IBM and BBC versions.)

TRACK: Users see a plan of a section of suburban railway with names blanked out, and a list of eight station names. Clues take the form

*DOCKS is a terminus.
COURT is between PARK and FERRY.
PARK is due north of GRAND.*

Users call up clues or fill in names at each turn, or else take a print out of the puzzle with a sufficiency of clues.

(IBM and BBC versions.)

SEESAW: Six outlines of children are shown at the top of the screen and a seesaw in the middle. One child is labelled with a weight. By convention weights are in multiples of 5 lbs in the range 50 lbs to 90 lbs. Users place the children on the seesaw by responding to the prompts “Who is on the left?” or “Who is on the right?” pressing appropriate initials. The program then moves the child to the seesaw, which goes up or down, at the same time printing a message of the form

*Ben weighs a little more than Alice.
Cora and Dick weigh less than Edward and Fiona.*

“A little more/less” implies a difference of 5 or 10 lbs, while “more/less” implies 15 or 20 lbs, and “a lot more/less” 25+ lbs. The children can be weighed in any combinations and any number of times in order to work out the missing weights.

(IBM version only.)

TRIP: Users see a blank diary page and a list of towns representing a tourist group’s itinerary. Clues take the form

*They went to Guildford before they went to Bath.
Their postcards of Brighton had a Bristol postmark.
They didn’t go to Oxford on a weekday.
Their visit to Cambridge was during the first three days.*

Users enter the towns against the days when they have seen enough information.

(BBC version only.)

CARFAX: Users see a schematic map of a crossroads. Clues take the form:

*If you approach the junction from the south, you have two choices.
Coming from the west you had better not turn left or you will be arrested.*

*King Street East is a one-way street.
You cannot drive south on Main Street South.*

Users mark the map with one-way arrows and no-entry signs when they have enough information.

(BBC version only.)

CUBE: Users see a multicoloured cube in a view which shows either two or three faces. They can 'roll' the cube, ie call up another random view of it, as often as they want in order to work out the placings of the six colours.

(BBC version only.)

Availability:

INVENNT is published by Research Design Associates, and is distributed in the UK by Wida Software, 2 Nicholas Gardens, London W5.

The remaining programs are unpublished. Any reader interested in using or delving into the IBM programs is invited to send a blank formatted 5 1/4" diskette to the authors, c/o COGITO, together with a pre-stamped return mailing envelope. We will put onto it TRACK and SEESAW in compiled form, together with a file of documentation and heavily annotated QuickBasic source code.

The BBC versions cannot be distributed at present, though we are ready to enter into correspondence.