Programmer's Introduction

By Ed van Zon

How it started

Dear User,

Congratulations! You've got a Macintosh computer to play with. At least, with me, that's what started the creation of the I Ching Connexion.
I play with my computer much like I used to play with Mecano: constructing something new out of it's basic building parts. I'm not really interested in the result (except that it has to work as intended), it's the screwing together that I like.
So, when back in 1988 I acquired my first Macintosh (a second hand Mac Plus), the big question was: what to build with it? Preferably something that someone else could be interested in, and that hadn't already been done. I decided a program that could find and present transcendental solutions of the China Labyrinth (as it was then called), would be sufficiently challenging for me and might be of interest to my friend Christiaan Freeling , who already had a collection of interesting solutions to this puzzle. Christiaan, by the way, was the inspiring force to evolve that program to the one you're looking at right now, in which the puzzle solving algorithm only plays a minor, albeit essential, part.

The Connexion: how it works

As explained in the 'Compiler's Introduction', the China Labyrinth puzzle consists of 64 hexagons, that all have different combinations of 'doors' on the sides: there's 1 hexagon with no doors, 6 have one door, etc.. In a transcendental solution every door is 'connected' to a door of a neighbouring hexagon, and every side without a door has no neighbour. (When you drop the second rule you get a 'compact solution'.)

The program searches for a transcendental solution using a simple backtracking technique: it puts the hexagons on a hexagonal grid one by one while observing the rules. Besides that it assumes the part put together so far is part of a solution until proven otherwise, in which case the hexagon placed last is removed from the grid and another one put in its place.
Note that the rules can only be upheld in a completely finished solution; in a partial solution (i.e. an 'unfinished' part of a solution) we have to relax the first rule: in a partial solution a door doesn't have to be connected (yet) to a door of a neighbour. It is clear however that to yield a solution eventually a

hexagon, with a corresponding door, must be put on the neighbouring position.

The algorithm (in short):

A, start:
The program starts by randomly picking a hexagon and placing it somewhere on the grid. Note that this one piece is part of every possible solution.
B, find a spot that requires a piece:
Since this is part of a solution, every side with an 'open' door will have to get a neighbouring hexagon. So, for the next step, just pick one of these sides. (*)
C, find candidates for that spot:
Determine which of the free pieces (the pieces that have not been put on the grid yet) could be put there as a neighbour, without breaking the (relaxed) rules. If the part put together so far is part of a solution (assumption), any such solution must have one of these free pieces, the candidates, on exactly that spot. (**)
D, try every candidate in turn:
So put one of the candidates on the designated spot, and continue with B again. If this doesn't yield a solution, replace this candidate with the next one and continue with B. Repeat this until all candidates have been tried. If none of the candidates yields a solution, then the part put together so far cannot be part of a solution, so we have to 'backtrack' even further.

(*) special cases finding a spot:
If there are no sides with an 'open' door left, a so-called group is finished. If, at the same time, all pieces have been put on the grid a solution is found! (To continue finding solutions, just go back to D and remove the last piece put.)
If there are still some free pieces, the program proceeds by randomly picking one of the free pieces and placing it somewhere on the grid to start building a new group. (The groups are considered not to interfere with one another.) Note that if the part consisting of the finished group(s) is part of a solution, putting this one piece also yields a part of any such solution. Continue with B.

(**) special case finding candidates:
If there are no candidates, the part put together so far cannot be part of a solution, and you'll have to go back to D and remove the last piece put.

Optimisations

All solutions the algorithm described above will present are unique , not considering rotations. And it will, given enough time, find every possible solution. You have to be patient though; it will take many millions of years to finish.
It might even take an unpractical amount of time to find a first solution, depending on how the first, say, 40 pieces are put. The computer can spend years trying to place the last 24, without there being a solution possible, given the arrangement of the first 40.
So I tried to speed things up a bit.

First, I made the program somewhat smarter in determining that a solution is not possible, given the part put together so far. E.g. if the part on the grid has more 'open' doors pointing upwards than there are downward doors on the free pieces, there clearly isn't a solution possible. The program is a bit more refined still, but this is the idea.

Second, in step B, finding a spot that requires a piece, the algorithm picks one randomly out of the several possibilities at a certain moment. Since any of the possibile spots will do for the algorithm, we might just as well take the spot that has the least number of candidates. This way, we keep this branch of the search tree at a minimum.

Third, a property of a transcendental solution is that the number of holes is the same as the number of groups. Note there are at least two groups (the hexagon with no doors, and the rest). So a solution must have at least two holes. I therefore tweaked the program (as suggested by Christiaan Freeling) in such a way that it would create holes early in the process. While this wouldn't speed things up if we were to find all solutions, it will speed up finding a first.
For this, note that a hole is surrounded by (at least) six pieces, that have doors at an angle of 120˚ (and no door in between). So, in step D, trying every candidate in turn, the candidate that has (the most of) these 120˚ angled doors is tried first. A consequence is that those kind of pieces are put on the grid early in the process, and relatively close to each other, so chances that a hole will be formed are made higher.

Fourth, also in step D, the candidates with a high number of doors are tried before the ones with few doors. Experience learns that, when most pieces are put on the grid, the chances of finding a solution is higher if the remaining free pieces have relatively few doors. So we want to get rid of the 'higher' pieces early in the process.

Even with the optimisations mentioned above, it happened fairly often that the program had put a part on the grid that couldn't be part of a solution,

but it was still frantically trying to complete it (and would continue trying for the next year or so). So, I decided (Christiaan approved, and not just on practical grounds as you can read in Interpretation: Connexion) to just start all over again if a solution wasn't found within a certain amount of steps (the action of putting a candidate on it's spot counts as one step).
This is rather drastic, for now the program isn't going to find all possible solutions anymore, but it assures that one will be found within a reasonable amount of time (depending on the hardware). Well, actually, there is no guarantee for this either, but statistics dictate one will be found reasonably quick most of the time.
For the number of solutions the program will find in an hour, there is an optimum in the number of steps the program is allowed to take before starting all over. This optimum is experimentally determined at 384.

The second, third and fourth optimisation still allow for a lot of randomness. There might be several spots with the least number of candidates; there might be several candidates with the highest number of doors. In such cases the program randomly picks one.
As you will note, the result is that the program never presents the same solution twice, and there is still an enormous variation in the solutions you'll get.

If you really want to, it is possible to adjust these optimisations, although we do not advise it for practical purposes. If you press âŒ˜-Y the 'Search Options' dialog will be presented to you. (Remember that a 'demand' is higher if that spot has less candidates, and a 'piece' is higher if it has more doors.)
What you might want to do is set 'place Pieces' at random. This can be done in the 'Preferences' dialog too.   On average it will take quite a bit longer to find a solution, but they are of a somewhat different form than the ones the program normally shows; e.g. solutions with two groups that have holes, which are very unlike to be found with the normal settings. Interesting as a screensaver, maybe.

The Octopuszle: how could it work

In the 'Compiler's Introduction' the Octopuszle is introduced as one of the generalisations based on the original China Labyrinth, and as one of the important motivations for creating the I Ching Connexion. Specifically, the observation that two persons independently found solutions to the Octopuszle that have the same diagonal complex, was the major incentive.

Now, how difficult is it to construct a solution with the same diagonal pattern as some given solution? I suggest you try it yourself.
Could a computer program achieve it, within reasonable time? Well, I

haven't written such a program, but it might be constructed analogous to the Connexion solving algorithm:

First note we're looking for a compact solution in a 16x16 square, with a fixed pattern of diagonals   coming from a known solution. So we know beforehand which spots require a piece (namely all 256); and 'open' doors will always be orthogonal ones (the blue lines).

The algorithm (very short):
A, find the spot with the least number of candidates.
B, find the candidates for that spot.
C, try every candidate in turn.

At the start, the corner positions have 4 candidates each, the positions on the border have 8 and the others all have 16 candidates. So, at the start, pick one of the corner spots.
Note that, at any time, the number of candidates in step B will never exceed 4, because there will always be a spot that has at least two of its sides determined (in other words, there will always be a 'corner' spot).
So an upper limit for the number of steps needed to complete the search for all   solutions is 4 to the 256th power.
Compare this with the upper limit for the I Ching Connexion: 8 to the 64th power.
Therefore, considering upper limits, it takes about 4^160 times longer to find all solutions for the Octopuszle (given a fixed diagonal complex) as the time needed for the I Ching Connexion. That is a factor of approximately 2.1 * 10^96.

How many solutions are there, given the diagonal pattern? Well, an upper limit is 16 * 16! ≈ 3.35 * 10^14. We feel that the number of solutions to the I Ching Connexion highly exceeds this number.

Summing up, it takes a hell of a lot more time to find less solutions.
So how long would it take to find one solution. Well, on average, more than a hell of a lot longer!
Suppose, we would find a solution for the Connexion at every clock-cycle of the computer (this is a ridiculously short time). Let's say our computer runs at 1000 MHz (never heard of one, at the moment).
It would then take more then 6.7 * 10^79 years to find one for the Octopuszle (with fixed diagonals), on average.

Even with the 'start-all-over' trick, we would be incredibly lucky to find a solution within 10 years.
On the other hand, we have been incredibly lucky finding the two solutons you can see in this program; i.e., to me it's a highly remarkable coincidence, to Christiaan it's the hand of Someone up there.

The natal hexagram display

The display of the natal hexagram is a bit simple: just the 'Earlier Heaven' and 'Later Heaven' hexagrams are displayed, with their controlling line inverted. Each line of the hexagram stands for a 6 or 9 year period (as explained in 'Intro Astro') and it is up to the user to divide the subject's life into these periods; the program doesn't help you with that.

It is also possible to 'Connect' the natal hexagrams although it is not clear what significance that would have. But if you can relate any meaning to a natal Connexion, you're welcome to do so.
You can always choose 'Hide Connexion' to return to the simple display.

Credits

This program wouldn't have seen daylight if it wasn't for the inspiration of Christiaan Freeling.
Textcompilation, texts (except the one you're reading now and balloontext) and all of the artwork are from his hands too.

The I Ching text in this program is basically the translation James Legge made (published in 1882, England) of the Imperial Edition of the I (published in 1715, China).

The link between the I Ching and astrology was suggested to us by Gerard Dijkman. The astrological data and calculation method are from 'The Astrology of I Ching' by Dr. W.K. Chu and W.A. Sherrill.

The I Ching Connexion is written using the THINK C development environment, so portions © Symantec Corporation.

Special thanks goes to Marco Piovanelli for his (free) Worldscript Aware Styled Text Engine (WASTE), which proved a considerable improvement over Apple's TextEdit.
WASTE text engine © 1993-1995 Marco Piovanelli.

I used quite a bit of sample code, bits and pieces I found on the Internet. Sometimes I modified things to serve my needs.
In no particular order, credits and my thanks go to:

F. Pottier for a set of (Pascal) routines to handle floating windows. Originally based on a set of C routines written by Patrick Doane.

Troy Gaul of
Infinity Systems for the Infinity Windoid WDEF © 1991-95 Infinity Systems.

Glenn R. Howes for the Flag Control CDEF.

Jim Stout for his Groupbox, Popup Menu and Date&Time CDEFs, and a set of routines to handle movable modal dialogs.

Harold Ekstrom for his SliderCDEF ©1993-1994.

Pete Gontier for a piece of asynchronous sound playing code.

Tony Myles for his SpriteWorld animation framework, and a set of handy dialog utility routines.

This program contains material from a version of the sample program "Chassis", which was originally created and copyrighted by Charles A. Hoffman.
Some of the "Chassis" code was used as a starting base to handle the text-windows of the I Ching Connexion. I have changed much and added a lot.

The Connexion document windows are of my own design & implementation, as is all the other code not credited for above.

Legal stuff

I Ching Connexion © 1995 Solar Software

Portions © by others (see Credits)

This software is distributed as shareware: if you like it please honor the shareware system and register by sending US$20 (or the equivalent in a major currency) to the author at the address below. In return you will receive a registration code which allows you to save and print I Ching Connexion documents.
Since cashing international cheques is very expensive, please send cash only. For the same reason I unfortunately cannot accept credit cards .

Please note that 50% of the contributions I receive will be donated to UNICEF, the United Nations Children's Emergency Fund.

Version history

version 1.0:
A bare China Labyrinth solving program.

version 2.0:
First public release

version 2.1:
- Added astro calculations (clock, natal) for the southern hemisphere. Before these were only applicable to the northern hemisphere.
- Reworked "Natal data" & "Clock settings" dialog. Hopefully, they're less confusing now.

version 2.2:
- Added a simple outcome display, so the program can be used for traditional divinations (that is: without a Connexion).
- Made the changing lines more visible, and clickable in the 'Hexagram Meaning' window.
- Switched to using the WASTE text engine, which brought along:
    · Drag and drop
    · Undo
    · Embedded pictures
    · Large texts (so Ta Chuan in two parts now, instead of four)
- Added Color menu.
- Added support for page navigation and text editing keys on extended keyboard.
- Several cosmetic changes, minor text corrections and small bug fixes.
- Optimised for PowerPC and 680x0 Macs (FAT version).

- and please note my new e-mail address: edvanzon@euronet.nl

How to reach me

If you experience problems with this software, or encounter bugs using it, I would like to hear about it. I might be able to fix it.
I'm open for any additional comment you would like to share about this program. Also, you can send the shareware fee to the address below.

You can reach me on this postal address:

Ed van Zon
Solar Software
P.O. Box 266
6700 AG   Wageningen
Netherlands.

or using the Internet:
edvanzon@euronet.nl

Have fun.

Ed van Zon.