Playing With Mazes

David B. Suits
Department of Philosophy
Rochester Institute of Technology
Rochester NY 14623

david.suits@rit.edu

1. What is a Maze?

A maze or labyrinth is a network of passages, usually intricate and confusing. What counts as intricate and confusing depends, of course, on the mind of the beholder. For most of this paper I shall be concerned with artificial beholders, namely, computers, and how algorithms and data structures can be specified for the automated construction and solving of mazes. For this reason, the mazes to be studied need not be particularly complex by human standards; I shall be investigating generalized principles which will apply to any degree of complexity of mazes — or, rather, of certain subclasses of mazes. Most of the presentation will be non-formal, as befits the subject matter.

The intersection of two passages will be called a **cell** (or **room**) of a maze such that there is a pathway from one passage to the other via that intersection. A maze is **planar** if it is constructable on a (two dimensional) surface such that passages do not cross except at such intersections. Non-planar mazes would be three (or higher) dimensional networks of passages and rooms.

The principles of maze construction and solution do not change for higher dimensional mazes, and so, for the sake of clarity and ease of drawing diagrams, I shall confine my attention to planar mazes only.

A passage may have a "cost" associated with it, perhaps in terms of the length of the passage, or in terms of hazards, or in terms of tiny trolls who threaten the traveler with unhappy consequences if some suitable tribute is not paid. Or sometimes even if it is paid. A system of roads in a city might be representable as a maze (though not necessarily planar). So might the playing field

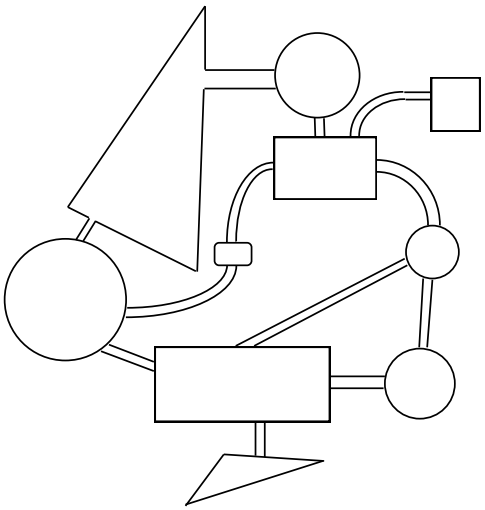


Figure 1. A fictitious maze in the courtyard of Mon, the fictitious emperor of the fictitious realm of D-D, described in a fictitious science fiction novel. The big circle is the peanut gallery.

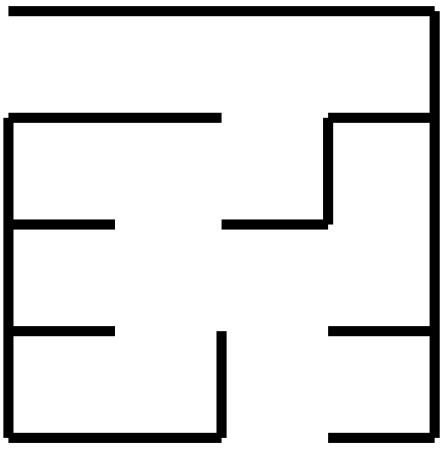


Figure 2. The floor plan of my grandparents' house, roughly as I conceived it as a child. (It bears little resemblance to the actual floor plan.)

of a pinball machine, the stacks in a library, a bureaucracy, a judicial system, a flow chart for a computer program, a crosswords puzzle, or the circulatory system of a door mouse.

The pictorial presentation of a maze might take different forms: the cells and passages might be explicitly drawn (figure 1); the passages might be of length zero, so that adjacent cells have **doors** and **walls** (or open and closed doors) (figure 2); or the cells might be of zero size, existing only conceptually at the intersections of passages (figure 3); finally, any maze may be represented abstractly as a **graph**, with nodes (vertices) and arcs (edges) (figure 4).

I shall be concerned in this paper only with mazes of the second type, which I shall call **regular floor plan (RFP) mazes**, and which are a subset of mazes representable as graphs. Somehow, RFP mazes seem to me to be "traditional" in a way the others are not; but this may be only a childhood prejudice. In any case, it will simplify some of the discussions; they represent a class of mazes of potentially great complexity; and they are easy to present on a computer screen or printer.

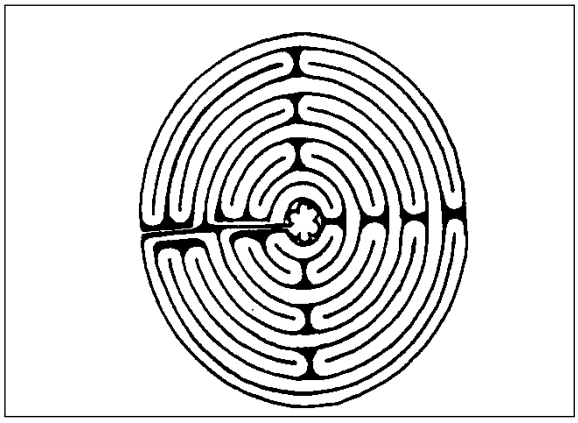


Figure 3. A real maze. A 13th century labyrinth at Chartres cathredral.

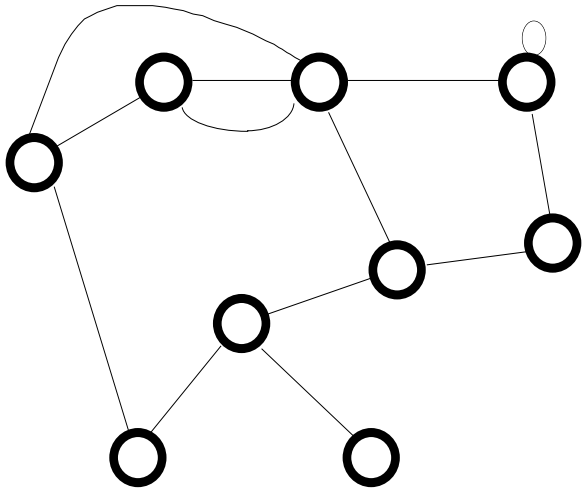


Figure 4. A real abstract maze.

The limitations of RFP mazes are obvious once they are compared with graphs. For example, from one of the nodes in figure 4 there is a passage leading back to the same node. This is not explicitly representable in RFP mazes (except by convention: a maze traveler may move from his present cell immediately to his present cell simply by staying put). Second, graphs may have any number of edges (passages) leading out of or into a node, whereas RFP mazes have no more than four passages (i.e., four immediately accessible neighbors). Third, a graph may be **directed**, such that the passages are one-way only. (RFP mazes might, correspondingly, have "one-way doors" between cells.) Since any maze, including RFP mazes, may be represented as graphs, various theorems from graph theory may be applied to the construction and solution of RFP mazes.

RFP mazes may be either **closed** or **open**. An open maze has an entrance from the outside and/or an exit to the outside, as shown in figure 5. In a closed maze, however, one of the cells might be designated the **start** cell, and one (or more) of the cells might be designated the **goal** cell, as in figure 6. Any open maze may be enlarged so as to become a closed maze which now includes the "entrance" cell and the "exit" cell (figure 7). Closed mazes being the more general variety, our attention will be focused primarily on closed mazes.

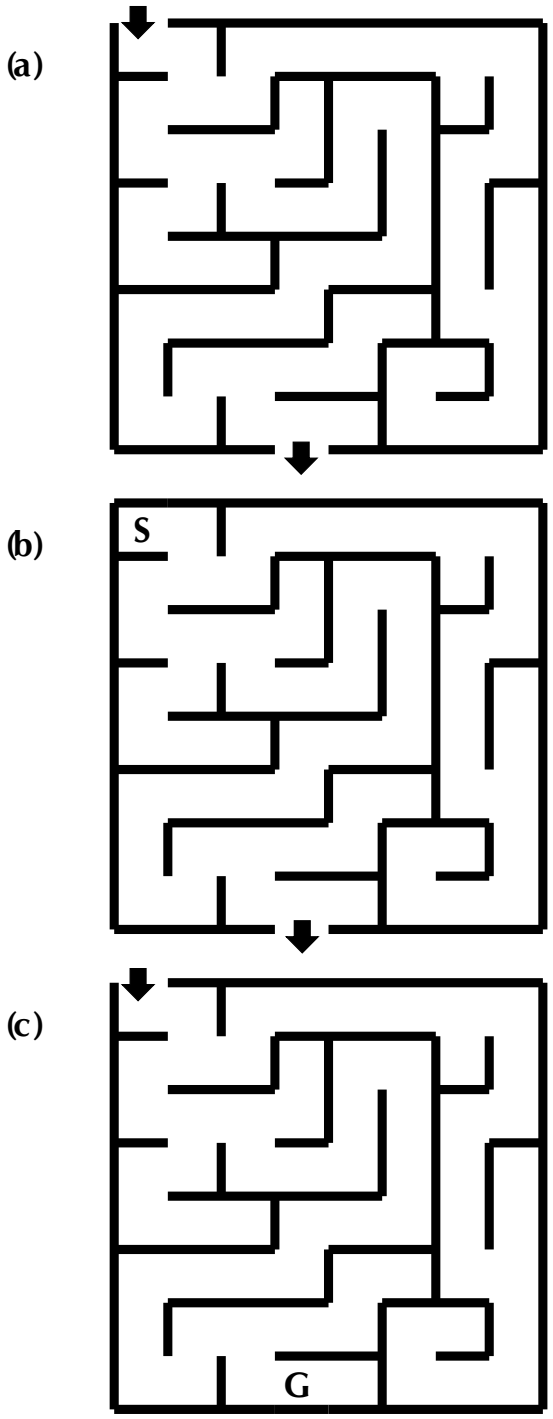


Figure 5. An open maze. (a) Find a path from the entrance to the exit. (b) Find a path from the start cell (inside the maze) to the exit. (I.e., escape from the maze.) (c) Find a path from the entrance to the goal cell (inside the maze). I.e., find the treasure (and possibly find your way back out).

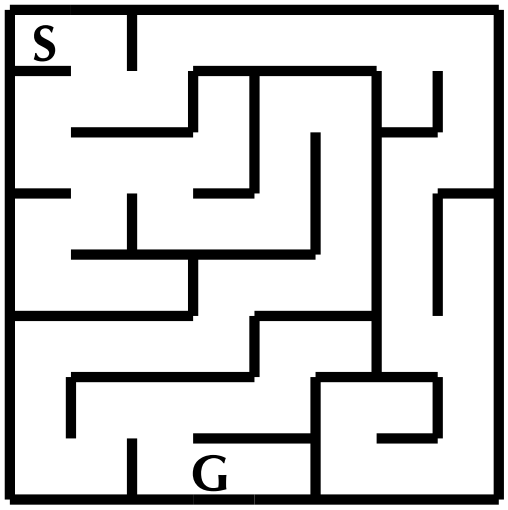


Figure 6. A closed RFP maze. Find a path from the interior start cell, S, to the interior goal cell, G.

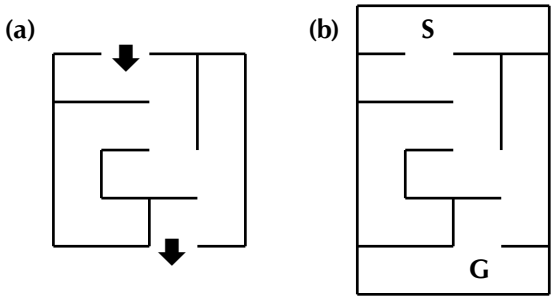


Figure 7. (a) An open maze. (b) A closed maze constructed by enlarging the open maze in (a).

2. Maze Construction

A maze is said to be **connected** if there is a path from any cell to any other cell. We want all mazes (which, henceforth, shall be only closed, RFP mazes) to be connected so that any two cells may be designated the start and goal cells with the guarantee that the maze is solvable. How can this guarantee be obtained? The algorithm to *solve* a maze — to find a path from the start cell to the goal cell — would be inefficient to construct a maze. For suppose we have some random maze, i.e., each side of each cell in the maze has randomly either a door or a wall, with the provision that neighboring cells have their adjacent sides the same type — both doors or both walls. (We will also assume in all of our discussions — unless otherwise indicated — that the exterior sides of the border cells of the maze are always to be walls. This is only another way of saying that the maze is closed.) Such a maze is of course easily constructed. Now, a maze solving algorithm will work only if there really is a solution path. Consequently, if the algorithm reports failure, we know that the maze is unconnected. In that case, we may remove a wall at random and run the algorithm again. And we continue removing walls until the algorithm reports success, which it will eventually do. But even given a solution path in the maze, we still have no guarantee that it is a connected maze, which requires a path from *any* start cell to *any* goal cell. We seek, therefore, a more efficient algorithm. The following are only some of the possible methods.

2.1 The Spanning Tree Algorithm

1. Choose the size (width and height) of the maze, and make it fully unconnected, i.e., no doors in the maze. Mark all cells as *unvisited*.
2. Choose a cell at random and call it the *present* cell, and mark it as visited (i.e., part of the tree).
3. For each neighbor (up to four of them) of the present cell, if it is marked as unvisited, mark it as *frontier*.
4. Choose any frontier cell in the maze at random and make it the present cell; connect it to any (random) neighbor marked visited. (Step 3 guarantees that every frontier cell will have at least one visited neighbor; and it is possible that repeated applications of step 3 might have created frontier cells with more than one visited neighbor.) If there are no frontier cells in the maze, go to step 6.
5. Mark the present cell as visited, and go to step 3.
6. The maze is now connected.

The spanning tree algorithm will generate a singly connected maze, that is, a maze with one and only one path from any cell to any other cell. (See figure 8.) Sometimes, however, it may be required to construct a maze with multiple paths between cells. In that case, take the connected maze and randomly remove a few walls (except for exterior walls). The resulting maze will be multiply connected.

2.2 Anderson's Algorithm

Peter Anderson (Dept. of Computer Science, Rochester Institute of Technology) suggested to me the following method of maze construction. It is basically the spanning tree algorithm, but its function is not to connect unconnected cells, but rather to "grow" a set of walls which, when complete, will result in a connected maze.

1. Begin with a maximally connected maze, i.e., a maze with no walls (except, of course, along the exterior of the maze).
2. Randomly pick any cell corner to which no wall is attached. If there is no such corner, stop; the maze is finished.
3. If possible, construct a wall from that corner to a randomly chosen neighboring corner, provided the neighboring corner is part of a wall. (I.e., connect a new wall to an already existing wall.)
4. Go to step 2.

Figure 9 shows a five by five cell maze in the process of construction according to Anderson's algorithm.

As in the spanning tree algorithm, Anderson's algorithm generates a singly connected maze. Multiple connections may be formed, as before, by randomly removing some walls. (An option is to add step 1.5: Randomly choose any door and make it a wall. It is possible — but not likely — that if that initial wall is not connected to the maze border, then the resulting maze might be multiply connected, namely, by having a complete path around the interior of the outside border of the maze.)

2.3 Manipulating a Connection Matrix

A connection matrix is a two dimensional array indicating which cells are connected to which other cells. Figure 10b, for example, is the connection matrix for the maze in figure 10a. An entry of "1" indicates a door between the cells whose numbers appear at the left column and the top row. Each cell is trivially connected to itself. Is there some way to manipulate a connection matrix so that it will eventually represent a connected maze? That is, are there properties of connection matrices which are necessary and sufficient for the representation of connected mazes? There certainly are a few necessary conditions which we may notice at once: (1) The matrix must be symmetrical about the main diagonal. (This is not necessary for mazes with one-way doors.) (2) Every row must have at least two "1" entries (i.e., each cell must be connected to at least one cell other than itself.) Hence, because of the symmetry, each column must have at least two "1" entries. (3) Some connections are impossible (for RFP mazes), as shown by the shaded areas in figure 11.

Unfortunately, those three conditions are not by themselves sufficient to guarantee a connected maze, and it seems unlikely to be able easily to deduce the missing conditions. That is, a random placement of "1" entries in the matrix according to the three conditions will not guarantee a connected maze.

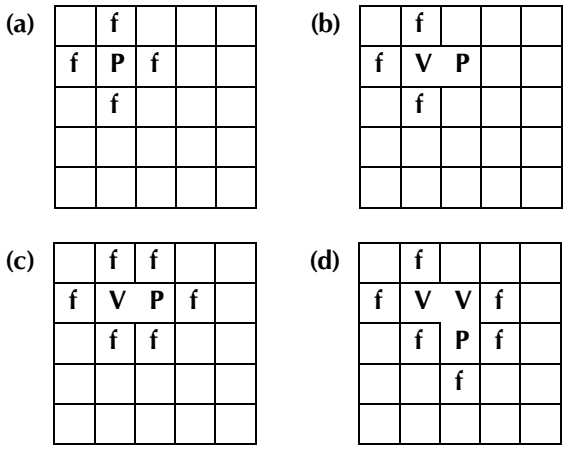


Figure 8. A maze being constructed by the Spanning Tree Algorithm. "P" represents the present cell, "f" indicates the frontier cells, and "V" represents visited cells, i.e., cells in the solution path. (a) After step 2. (b) After the first time through step 4. (c) After the second time through step 3. (d) After the third time through step 3.

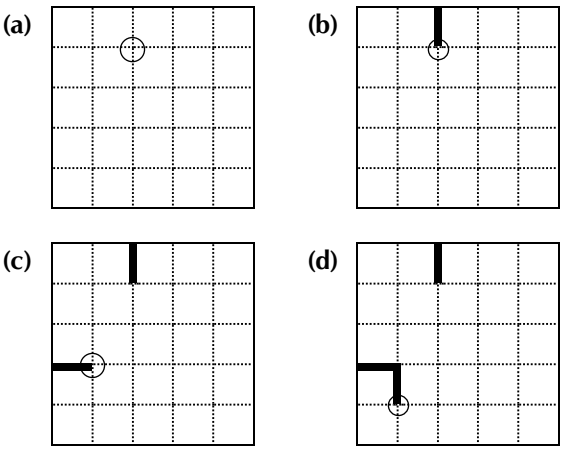
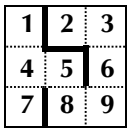


Figure 9. A maze being constructed by Anderson's algorithm. (a) After the first time through step 2. (b) After the first time through step 3. (c) After the second time through step 3. (d) After the third time through step 3.



(b)

	1	2	3	4	5	6	7	8	9
1	1	0	0	1	0	0	0	0	0
2	0	1	1	0	0	0	0	0	0
3	0	1	1	0	0	1	0	0	0
4	1	0	0	1	1	0	1	0	0
5	0	0	0	1	1	0	0	1	0
6	0	0	1	0	0	1	0	0	1
7	0	0	0	1	0	0	1	0	0
8	0	0	0	0	1	0	0	1	1
9	0	0	0	0	0	1	0	1	1

Figure 10. (a) A 3×3 maze. (b) The connection matrix for the maze in (a).

One reason for that is that each entry in the matrix represents where a maze traveler could go *in one move*, and the three conditions above tell us only that a traveler could go *somewhere* in one move. What we need, then, is a matrix telling us where a traveler could go in n moves, where n is the longest of all the longest paths from each cell to every other cell, which is to say that n is equal to one less than the number of cells in the maze.

More information is possible if the matrix is multiplied by itself. Let M^1 be the name of the connection matrix, and let $M^2 = M^1 \bullet M^1$. Then each entry in M^2 will indicate the number of paths of *two* moves from a given cell to all others. Here's why: Figure 12 shows the multiplication of two matrices, A and B, yielding the new matrix C. Notice that the subscripts on each element of each maze indicate two cells which are being examined, and the value of the element is the number of ways to get from the first cell (a_{ij}) to the second (b_{ij}) in one step (that is, both A and B represent M^1). Let's take a look at one of the entries in C, say c_{12}. The element a_{11} is the number of ways to get from cell 1 to cell 1 in one step (and so its value is 1 — i.e., the maze traveler might stay put, as mentioned before). The element b_{12} is the number of ways to get from cell 1 to cell 2 in one step (and the value is 1, because cell 1 is connected to cell 2). We must add to that the product of a_{12} (the number of ways to get from cell 1 to cell 2 in one step) with b_{22} (the number of ways to get from cell 2 to cell 2 in one step) and the product of a_{13} (the number of ways to get from cell 1 to cell 3 in one step) with b_{32} (the number of ways to get from cell 3 to cell 2 in one step). The result is therefore the number of ways to get from cell 1 to cell 2 in two steps. An inspection of the M^2 matrix for the maze in figure 13 will serve to illustrate the result.

Now if we multiply M^2 by M^1, we shall get M^3, representing the number of ways to get from each cell to all others in three steps. Continuing in this fashion, we will eventually obtain M^8, which will represent the number of ways to get from each cell of every other cell in 8 steps. For the maze in figure 13, there are 9 cells in the maze, and so 8 steps ought to be sufficient to get from any cell to any other cell, provided the maze is connected. Consequently, if every entry in M^8 is non-zero, the maze is connected, as is the case for the maze in figure 13.

(a)

1	**2**	**3**
4	**5**	**6**
7	**8**	**9**

(b)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	1								
2		1							
3			1						
4				1					
5					1				
6						1			
7							1		
8								1	
9									1

Figure 11. (a) A 3×3 maze; the interior walls have not yet been added. (b) The beginning of a connection matrix for the maze in (a). Shaded areas represent impossible connections for RFP mazes.

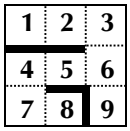
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \bullet \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$

$$\begin{bmatrix} c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & c_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & c_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & c_{23} = a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ c_{31} = a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & c_{32} = a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & c_{33} = a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

Figure 12. The general form of matrix multiplication.

If, however, M^8 tells us that the maze is not connected (see the example in figure 14), then we can do something reasonable to make it connected. An unconnected maze is one wherein there are two or more regions (of one or more cells each) which are inaccessible from each other. Choose any two *neighboring* cells in the maze whose entry in M^8 is zero. These will be neighbors on either side of the border between two such mutually inaccessible regions. Change the maze so that the two cells (and hence the two previously unconnected regions) are now connected. Compute M^8 again. Continue the process until every entry in M^8 is non-zero. Actually, it might not be necessary to compute M^8 if all entries in some M^i, $i < 8$, are non-zero. (See M^7 in figure 13d.) To generalize, if n is the number of cells in a maze, then the maze is connected if there is some $i \le n-1$ such that all of the entries in M^i are non-zero. Such an M^i we will call the **complete connection matrix**

(a)



(b)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	1	1	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0	0	0
3	0	1	1	0	0	1	0	0	0
4	0	0	0	1	1	0	1	0	0
5	0	0	0	1	1	1	0	0	0
6	0	0	1	0	1	1	0	0	1
7	0	0	0	1	0	0	1	1	0
8	0	0	0	0	0	0	1	1	0
9	0	0	0	0	0	1	0	0	1

(c)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	2	2	1	0	0	0	0	0	0
2	2	3	2	0	0	1	0	0	0
3	1	2	3	0	1	2	0	0	1
4	0	0	0	3	2	1	2	1	0
5	0	0	1	2	3	2	1	0	1
6	0	1	2	1	2	4	0	0	2
7	0	0	0	2	1	0	3	2	0
8	0	0	0	1	0	0	2	2	0
9	0	0	1	0	1	2	0	0	2

(d)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	127	197	196	28	77	161	7	1	70
2	197	323	358	84	189	343	29	7	161
3	196	358	470	190	350	547	84	28	273
4	28	84	190	393	385	350	330	189	162
5	77	189	350	385	477	548	266	134	273
6	161	343	547	350	548	750	189	77	386
7	7	29	84	330	266	189	316	196	77
8	1	7	28	189	134	77	196	127	27
9	70	161	273	162	273	386	77	27	204

(e)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	324	520	554	112	266	504	36	8	231
2	520	878	1024	302	616	1051	120	36	504
3	554	1024	1375	624	1087	1640	320	112	820
4	112	302	624	1108	1128	1087	912	519	512
5	266	616	1087	1128	1410	1648	785	400	821
6	5041	1051	1640	1087	1648	2231	616	266	1136
7	36	120	302	912	785	616	842	512	266
8	8	36	112	519	400	266	512	323	104
9	231	504	820	512	821	1136	266	104	590

Figure 13. (a) A 3×3 maze. (b) M^1 for the maze. (c) M^2 for the maze. (d) M^7 for the maze. (e) M^8 for the maze.

and denote by M^c. Matrices derived from M^c will be used later in Section 4.

3. Solving Mazes

Once a maze is known to be connected, there are probably countless methods of automating a solution. I shall present some simple methods and some complex methods.

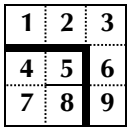
3.1 Random Walk

Beginning in the start cell, a random walk from cell to cell could eventually lead to the goal cell. If directions are chosen pseudo-randomly such that the distribution of choices in the long run covers all possibilities, then of course the goal cell will eventually be reached. Obviously, though, this method is better thought of as a non-method.

3.2 The Left- (or Right-) Hand Walk

Starting at the start cell, keep your left (or right) hand touching a wall. Walk. Eventually (with certain kinds of mazes) you will reach the goal cell. The maze in figure 13 is amenable to this solution, but the maze in figure 15 is not. The reason is that the maze in figure 15 has multiple paths between some cells (it is ***multiply connected***). Suppose cell 5 is the start cell and cell 9 is the goal cell. A left-hand walk starting in cell 5 and facing east (or a right-hand walk starting in cell 5 facing west) will consist of an infinite loop

(a)



(b)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	1	1	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0	0	0
3	0	1	1	0	0	1	0	0	0
4	0	0	0	1	1	0	1	0	0
5	0	0	0	1	1	0	0	0	0
6	0	0	1	0	0	1	0	0	1
7	0	0	0	1	0	0	1	1	0
8	0	0	0	0	0	0	1	1	0
9	0	0	0	0	0	1	0	0	1

(c)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	2	2	1	0	0	0	0	0	0
2	2	3	2	0	0	1	0	0	0
3	1	2	3	0	0	2	0	0	1
4	0	0	0	3	2	0	2	1	0
5	0	0	0	2	0	0	1	0	0
6	0	1	2	0	0	3	0	0	2
7	0	0	0	2	1	0	3	2	0
8	0	0	0	1	0	0	2	2	0
9	0	0	1	0	0	2	0	0	2

(d)

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	323	512	517	0	0	384	0	0	195
2	512	840	896	0	0	712	0	0	384
3	517	896	1035	0	0	6	0	0	3
4	0	0	0	7	5	0	6	3	0
5	0	0	0	5	4	0	3	1	0
6	1	3	6	0	0	7	0	0	5
7	0	0	0	6	3	0	7	5	0
8	0	0	0	3	1	0	5	4	0
9	0	1	3	0	0	5	0	0	4

Figure 14. (a) The same maze as in figure 13a, but with a wall between cells 5 and 6, making the maze unconnected. (b) M^1 for the maze. (c) M^2 for the maze. (d) M^8 for the maze.

traversing cells 5, 6, 3, 2, 1 and 4. Cells 7, 8 and 9 will never be visited. A left-hand walk beginning at cell 5 but facing west, or a right-hand walk beginning in cell 5 facing east, will, however, find a solution.

An improvement on this method is to keep track of cells visited. If you return to a visited cell, there is a loop, and to break out of it, switch from a left- (right-) hand walk to a right- (left-) hand walk. This method will solve the maze in figure 15.

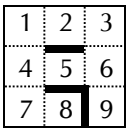


Figure 15. A multiply connected maze which cannot necessarily be solved by a left-hand or right-hand walk.

But both the simple and improved versions of this method will fail to solve mazes where the goal cell is a "King's chamber", i.e., a cell none of chose corners is part of a wall. Such a maze is shown in figure 16.

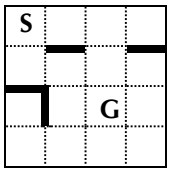


Figure 16. The left-hand or right-hand walk does not guarantee a solution to a maze which the goal cell is a "King's Chamber".

3.3 Least Recently Used Walk

A simple (but not speedy) algorithm which guarantees the solution of any maze extends the idea of marking the cells in order to take note of revisits. But in this case it is the doors, and not the cells themselves, which are marked, and the goal cell is (eventually) discovered by exiting whatever cell you are in through the least recently used door.

1. Mark all doors with "0".
2. Set $i \leftarrow 0$.
3. If the present cell is the goal cell, stop.
4. $i \leftarrow i + 1$.
5. Choose the door with the lowest number. (If more than one door has the same lowest number, randomly choose between them.)
6. Mark the door as *i*, and exit through that door.
7. In the neighboring cell, mark the door through which you came as *i*.
8. Go to step 3.

In case the maze is so large that an automated solution requires numbers larger than the representative power of the machine being used, it may help to realize that it is not necessary to increment *i* in the manner given in the algorithm above. Rather, *i* need be set only to one more than the highest number in the present cell. So for step 7 we could substitute:

7′. In the neighboring cell, set *i* to the highest number in the cell + 1, and mark the door through which you came as *i*.

Even smaller numbers can be maintained by any simple scheme (not to be detailed here) which always renumbers the doors of a cell just entered from 0 (least recently used) incrementally to most recently used (i.e., the door you just came through). I mention this possibility only because such simplifying measures were necessary for a version of this algorithm which I implemented on a small, programmable calculator (a TI-58, circa 1977).

The main virtue of the Least Recently Used algorithm is that its memory requirements are minimal. The main vice of the algorithm is that since many cells might be visited many times, its time requirements can be huge. Another virtue of the method is that once a path from the start to the goal has been achieved, a path back to the start cell is available by going through doors with the highest value (not counting the door through which you entered the present cell). But another vice of the algorithm is that it does not guarantee to find the *shortest* path to the goal cell; indeed, it does not necessarily find all paths to the goal cell, so that even if you were able to pick the shortest of the paths traversed, there might be even shorter paths not yet investigated.

Pruning algorithms may be added to this (and to most any other) solution algorithm. For example, a cul-de-sac, once discovered, might be specially marked (say, with a negative number) so

that it is henceforth no longer used. By this means, a tunnel (a string of one or more cells with exactly two doors) which leads into a cul-de-sac will itself be incrementally closed off. That having been done, a solution path can be more efficient. Sometimes a loop can be discovered and marked so that it will not be traversed again. But this requires significantly more memory resources (and a little bit more time en route) to manage, and so I will not discuss such methods here. (See, for example, Allen 1979.)

3.4 Breadth-First Search

The breadth-first search algorithm labels cells (not their doors), searching from the start cell to all its immediate neighbors. If the goal cell is not found, the search is performed outward to the neighbors of each of the neighbors of the start cell; and so on until the goal cell is found. The algorithm keeps track of which cells are immediate neighbors of the start cell, and which cells are neighbors of those immediate neighbors, etc., by labeling each set of cells (neighbors, then neighbors of neighbors, and so on) with higher and higher numbers.

1. Label the start cell as 0.
2. $i \leftarrow 0$.
3. For each cell labeled i, label all unlabeled adjacent cells with $i + 1$. (If there are no such adjacent cells, stop; the maze is unconnected.)
4. If any of the newly labeled cells is the goal cell, stop; a solution path has been found.
5. $i \leftarrow i + 1$.
6. Go to step 3.

The results of a breadth-first search are shown in figure 17. Notice that a solution path has been found. (It is possible that in a multiply connected maze multiple solution paths will be marked out.) Unfortunately, such a search is best performed in parallel; for a poor traveler trying to find his way through the maze, the breadth first search method would involve so much backtracking as to make the algorithm very tedious (although perhaps not as bad as the Least Recently Used Algorithm). Moreover, additional memory is required to keep track of what "level" of the search one is on, and, for each given cell at the previous level, whether its immediate neighbors have been labeled at the new level. In addition, the cells, once having been labeled, do not provide a very efficient means of traveling that same path again without going through another breadth-first search. The labeling does, however, provide an excellent means of returning to the start cell from the goal cell: simply move to the neighbor cell with the lowest number. As a bonus, the number in a cell also tells you how many cells remain between your present position and the start cell. (Since breadth-first search might mark out multiple solution paths, there could also be multiple return paths.)

The breadth-first search method, if started at the goal cell in search of the start cell, will provide a clear (and shortest) path from the start to the goal. (Just move to that neighbor with the lower number.) And the label in a cell will indicate the distance to the goal. Of course, retracing one's path will involve as much backtracking as finding the goal cell did in the original version.

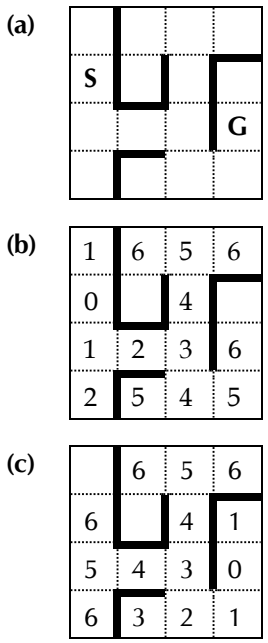


Figure 17. (a) A 4×4 maze with start and goal cells. (b) The final labelling generated by breadth-first search. (c) A breadth-first search started from the goal cell back to the start cell.

3.5 Tremaux's Algorithm

Tremaux's algorithm (see Even, 1979, pp. 53f) is a depth-first search used to visit all cells, terminating in the original cell after having examined all doors once in each direction. (The Hopcroft-Tarjan version of depth first search is described in Even 1979, p. 56. Variations on Tremaux's algorithm are given in Even 1979, pp. 66f.)

1. Make all doors unmarked.
2. Choose some cell to be *p*, the present cell.
3. If there are no unmarked doors, go to step 7.
4. Choose any unmarked door, mark it "E", and go through to the neighboring cell.
5. If this neighbor has any marked doors (i.e., if the cell has been visited before) mark the door back to *p* with "E", go back to *p*, and go to step 3.
6. This neighbor has not been visited before. Mark the door back to *p* with "R", assign this cell to *p*, and go to step 3.
7. If there is no door marked "R", stop; we are back in the start cell, and the whole maze has been visited.
8. Go through the door marked "R", assign this new cell to *p*, and go to step 3.

Figure 18 shows the progress of this algorithm. If it is necessary only to find *some* path to the goal cell (rather than to visit every cell), then the following step could be substituted for step 6:

6'. Mark the door back to *p* with "R" and assign this cell to *p*. If *p* is the goal cell, stop. Otherwise, go to step 3.

Tremaux's algorithm will find the goal cell (provided, of course, that the maze is connected), and in addition it will provide a clear return path to the start cell. It also has the advantage of marking each door only once, unlike the Least Recently Used algorithm. However, it does not provide any means of repeating the solution path, except by once again solving the maze. But this can easily be remedied. During the retracing, use a counter, initialized to zero, and incremented by one each time you follow the "R" marks to a neighboring cell; replace the "E" on the door through which you just came with the value of the counter. Now a permanent path is given from the start to the goal simply by following the numbered doors. What's more, the numbers on the doors will indicate the number of steps necessary to get to the goal cell. Unfortunately, if there are any "cycles" in the maze — if, that is, it is multiply connected — the solution path is not guaranteed to be the shortest.

Another variation on the original algorithm is to include a test for the goal cell, at which time, instead of halting or simply retracing our steps (even with a counter), we continue the search, this time with the help of a counter. New cells found are incrementally marked, and during backtracking we decrement the counter. We will have to revisit some cells in order to mark them with counters. But when the process is complete — when we are back in the start cell after having visited all cells (some twice) — then each cell will have exactly one door marked with a number indicating the number of steps (through that door) from the start to the goal. (I suspect that this path will also be the shortest path, but a proof of that is, as they say, left up to the reader.)

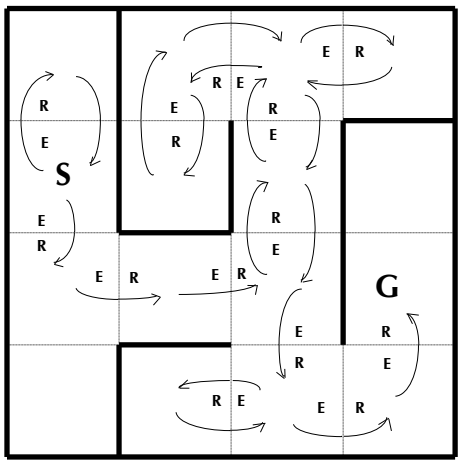


Figure 18. Tremaux's algorithm at work.

3.6 The Cheese Algorithm

All the previous methods of solving a maze relied on a maze traveler's own discoveries during movement though the maze. I suggested earlier in the discussion of the breadth-first search algorithm that if the search were reversed so that investigation proceeded outward from the goal cell, rather than outward from the start cell, then the maze traveler would have a clearly marked path to the goal cell. But how, in terms of a traveler in a maze, are we to conceive of such a backwards search from the goal? If we already knew where the goal cell was, we wouldn't have to search at all.

Suppose, however, we present the maze problem as a simulation of two events taking place simultaneously: the first is the traveler, trying to find a solution path as usual. The second is some kind of information from the goal cell being broadcast outwards. Imagine, then, that in the goal cell is a piece of cheese — perhaps Limburger cheese — the odor of which gradually permeates the entire maze. Naturally, the strength of the odor in any given cell will be proportional to the

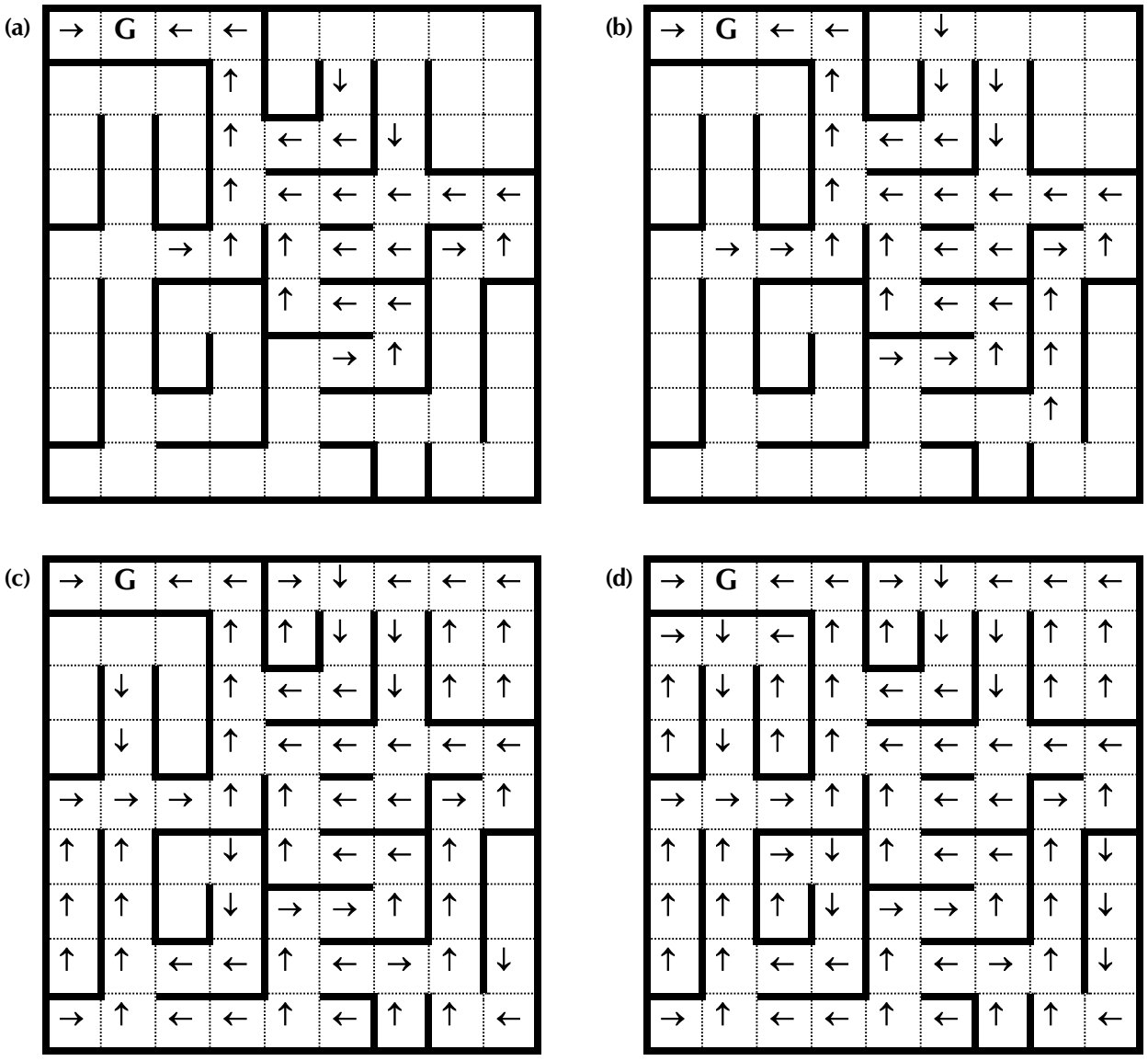


Figure 19. TAR spreading throughout a maze. The cheese is in the goal cell (G). Although the TAR for each cell changed over time, the direction of greatest TAR for each cell changed only rarely. (Arrows indicate, for each cell, the direction of greatest aroma.)

distance of that cell from the cheese itself. The maze traveler (which we might as well concede is a rat) need only stay put in the start cell (or, if the rat is smart enough, it might initiate one of the earlier search algorithms) until it smells the cheese, at which time it follows the rule: proceed in the direction of strongest cheese odor. And as cells are gone through, they may be marked so that retracing the path back to the start cell is possible, and so that the path from the start to the goal may be followed at any future time without again having to search.

The strength of the cheese smell can be simulated by attaching a number (say between 0 and 1) to each cell. Let us call such a number the *Total Aroma Rating*, or *TAR*. ("TAR" is "RAT" going backwards.) The cheese cell will have a constant TAR of 1, and all other cells will begin with a TAR of 0. After each unit of simulated time, the strength of the odor in a given cell will be the result of the aroma the cell already has, minus what it loses to all its immediate neighbors, plus whatever it gains from its neighbors. For the sake of simplicity, we may say that the TAR of a cell will be equal to the average of its own TAR plus its neighbors' TARs.

Although spreading TAR is similar to a breadth-first search from the goal to the start, there are some differences. TARs change, whereas in the breadth-first search, a cell once labeled retains that label. And if there are cycles in the maze, odor will spread along two paths and then tend to disturb each other as they join up later. The question is, will the rat be able to find the cheese simply by following increasing TARs?

The answer is, almost obviously, Yes. But instead of constructing a formal proof, a simulation was employed. Figure 19 shows the 9 by 9 cell test maze after a number of iterations as the TAR activation spreads. A number of such runs were made, with the expected results.

4. More on Connection Matrices

A disadvantage of all the maze solving algorithms is that none will mark out a path from any cell to any other cell. That is, given the same set of doors and walls, but a different start or goal cell, the entire algorithm will have to be run anew.

In section 2.2, M^c was called the complete connection matrix, a maze's connection to the ith power, where $1 \le i \le n-1$ (where n is the number of cells in the maze; hence, $n-1$ is the longest possible path in the maze) such that all entries in M^i are non-zero. Not only does M^c tell us whether the maze is connected, but it also gives us a basis for understanding *how* it is connected. Specifically, what we are looking for is a matrix which gives us the minimum distance from any cell to any other cell. Let M^d be such a *minimum distance matrix*. It will display in matrix form the structure of walls and doors of the maze. Such information will, in turn, provide us with a means for calculating the path from any start cell to any goal cell. That is, from M^d we will be able to construct M^{sg}, the ***solution matrix*** for the goal cell g.

4.1 Constructing M^d, the Minimum Distance Matrix

Repeatedly running one of the maze solving algorithms (with suitable enhancements) may provide us with M^d. But given M^1, M^d will be simple to calculate as we are constructing M^c. Once an entry in some M^i becomes non-zero, we know how many steps (namely, i) there are between the cell in the given column and the cell in the given row for that entry. Thus, i becomes the value for that entry in M^d. That is, for each of M^i, $i = 2 \ldots c$, all non-zero entries in M^i which were 0 in M^{i-1} are assigned the value of i in M^d (excluding M^d_{jj} entries — that is, the entries in the main diagonal — which are assigned 0 in M^d, since there are minimally 0 steps between any cell and itself). Figure 20 repeats the maze shown earlier in figure 15, and gives both M^c and M^d.

4.2 Constructing M^s, the Solution Matrix

Suppose, for the maze in figure 20a, that cell 1 is the goal cell. We wish to know how best to get to that cell from wherever we are. Suppose we are in cell 5. Consulting M^d we discover that cell 1 is two cells away. But in what direction? It is easy to find out. From cell 5 we can move only to cell 4 or to cell 6. (These are the entries in M^d in figure 20c indicating minimal distances of 1 from cell 5.) If we were to move to cell 4, then cell 1, according to M^d, would be one step away; if we were to move to cell 6, then cell 1 would be 3 steps away. Clearly, then, we get closer to cell 1 from cell 5 by

moving to cell 4. So the process for calculating M^{sg} for any g in a maze with n cells is as follows:

1. $M^{sg}_{ij} \leftarrow -1$ for all i, j (–1 simply indicates an invalid entry).
2. For $i = 1 \ldots n$, $i \neq g$
   For each k, $k = 1 \ldots n$ such that $M^{d}_{ik} = 1$ and $M^{d}_{kg} < M^{d}_{ig}$
   $M^{s}_{ik} \leftarrow M^{d}_{ig}$.
3. $M^{s}_{gg} \leftarrow 0$.

(a)

1	2	3
4	5	6
7	8	9

(b)

	1	2	3	4	5	6	7	8	9
1	56	47	35	63	40	33	35	15	10
2	47	51	47	40	32	40	17	5	16
3	35	47	56	33	40	62	10	2	30
4	63	40	33	91	64	45	61	30	17
5	40	32	40	64	61	63	35	15	30
6	33	40	62	45	63	86	18	5	46
7	35	17	10	61	35	18	51	31	5
8	15	5	2	30	15	5	31	21	1
9	10	16	30	17	30	46	5	1	26

(c)

	1	2	3	4	5	6	7	8	9
1	0	1	2	1	2	3	2	3	4
2	1	0	1	2	3	2	3	4	3
3	2	1	0	3	2	1	4	5	2
4	1	2	3	0	1	2	1	2	3
5	2	3	2	1	0	1	2	3	2
6	3	2	1	2	1	0	3	4	1
7	2	3	4	1	2	3	0	1	4
8	3	4	4	2	3	4	1	0	5
9	4	3	2	3	2	1	4	5	0

Figure 20. (a) A 3×3 maze. (b) $M^c = M^5$ for the maze. (c) M^d for the maze.

Figure 21 shows solution matrices for g = 1, 2 and 3.

Notice that each solution matrix is equivalent to the results of a breadth-first search initiated from the given goal cell outwards to the rest of the maze. Since the structure of a maze is known once M^1 is known, it is then a question of implementation-dependent efficiency whether to engage in matrix multiplication or whether to simulate breadth-first searches in order to create any (or all) desired solution matrices.

Although the solution matrix is perhaps not very useful for the ordinary solve-a-maze puzzle, it might be useful in certain variations. Suppose, for example, that the rat is looking for the cheese placed somewhere in a familiar maze. And suppose the cheese has legs with which to move away from the rat. OK, if that's supposing too much, then we may instead imagine a computerized demon trying to catch a human player in a maze game. As long as the demon knows the identity to the cell its prey is presently in, it can consult the proper M^s and make the most efficient move towards it. In effect, the demon faces the problem of a continuous automated solution, where the start cell and goal cell are changing in the same maze. Either the specific M^s is calculated as needed, or else all of them are calculated, stored and consulted later on as required.

4.3 Cautions

But let's be clear about what has been claimed for these matrices. If you were put at the entrance (i.e., in a start cell) of an unfamiliar maze and told to go find the treasure and return, then there would be nothing for it but to use a suitable maze solving algorithm; after all, you do not yet know the structure of the maze; and not only do you not know how to get to the goal cell, you do not know its identity. However, once given the structure of the maze — after having examined it on your first run — you can construct the connection matrix, and hence M^c, M^d and M^s. At that point you can efficiently retrieve any further treasure in the maze merely by being told the identity of the cell it is in. Suppose, however, that although you have M^d, you know only that there is a treasure somewhere in the maze. Even though in your search for the treasure you can now be more efficient than a traveler who knows nothing about the structure of the maze, you will never-

theless have to employ some search procedure or other. That is, you know how to get efficiently from any cell to any other cell; but since you do not know the identity of the goal cell, you will have to search for it. That is, you face a traveling salesman's problem: how shall you organize your travels along alternative but known routes in order to cover all the territory while incurring the least cost? (Cost, in this case, is distance. The least cost will be incurred for the least backtracking.) It is a restricted traveling salesman's problem in that the start cell is stipulated. I have a gut feeling that some sort of matrix manipulation might straightforwardly solve this problem (would that make me famous?), but I have no suggestions to offer at this time. (Gut feelings are not very reliable anyway.)

5. A Neural Net Maze Solver

The solution methods I have discussed involve putting various identifying marks in the cells of the maze (or, what comes to the same thing, "marking" a memory image of the maze). The memory — the record — which helps guide you through the maze is imprinted upon the maze (or a copy of the maze) itself. In order to determine what to do next, at any given position, you need only consult the marks in the present cell and apply an appropriate rule. If there is an independent memory in use, it is minimal: perhaps a counter, for example.

Imagine, however, that you are trying to move through a maze without the benefit of being able to leave identifying clues (or clews). Your own memory serves only as a more or less (un)reliable "feel" — a sense of context, perhaps, which grows with experience of the maze. That is, as you move through the maze you more and more confidently identify patterns of kinds of cells. You may learn, for example, that there is one section of the maze with a very long corridor, each cell of which has a door to the right leading to a cul-de-sac. If you have knowledge of such a corridor, then you might recognize it whenever you move through it, and this identification might serve adequately as a means of orientation. Such a memory is a behavior pattern.

(a)

	1	2	3	4	5	6	7	8	9
1	0	1	2	1	2	3	2	3	4
2	1	0	1	2	3	2	3	4	3
3	2	1	0	3	2	1	4	5	2
4	1	2	3	0	1	2	1	2	3
5	2	3	2	1	0	1	2	3	2
6	3	2	1	2	1	0	3	4	1
7	2	3	4	1	2	3	0	1	4
8	3	4	4	2	3	4	1	0	5
9	4	3	2	3	2	1	4	5	0

(b)

	1	2	3	4	5	6	7	8	9
1	0	1	2	1	2	3	2	3	4
2	1	0	1	2	3	2	3	4	3
3	2	1	0	3	2	1	4	5	2
4	1	2	3	0	1	2	1	2	3
5	2	3	2	1	0	1	2	3	2
6	3	2	1	2	1	0	3	4	1
7	2	3	4	1	2	3	0	1	4
8	3	4	4	2	3	4	1	0	5
9	4	3	2	3	2	1	4	5	0

(c)

	1	2	3	4	5	6	7	8	9
1	0	1	2	1	2	3	2	3	4
2	1	0	1	2	3	2	3	4	3
3	2	1	0	3	2	1	4	5	2
4	1	2	3	0	1	2	1	2	3
5	2	3	2	1	0	1	2	3	2
6	3	2	1	2	1	0	3	4	1
7	2	3	4	1	2	3	0	1	4
8	3	4	4	2	3	4	1	0	5
9	4	3	2	3	2	1	4	5	0

Figure 21. (a) M^{s1} for the maze in fig. 20. (b) M^{s2} (c) M^{s3}.

The problem now in solving a previously learned maze under the constraint of not being able to leave marks in the cells is to first orient oneself. Once oriented, it is a matter of following one's internalized "feel" in order to remain oriented.

This is usually inherently serial. One might not clearly anticipate the proper moves in advance; that is, you might not be able to recall, in advance, that at a certain point you will have to turn right. Rather, you will simply wait until the pattern seems to emerge on its own in response to the present context.

An artificial neural net embodies at least some of the properties we need in order to create such a memory. Neural nets do not store explicit memory images; rather, they contain a network of interactions which, given the proper stimulation, will *recreate* certain kinds of responses. We may say that the neural net's memory is its ability to engage in such recreations.

But most research on artificial neural networks has not focused on the time domain, whereas I am searching for a neural net architecture which will tend to serialize its recreations. I propose a sketch of a possible neural network to solve mazes. It is based roughly on an idea for serial learning proposed by Jordon (1986) and later used by Todd (1989). But whereas Jordon's network learned by means of back-propagation (wherein mistakes made by the net are corrected by an all-knowing teacher), my proposal is for a network which teaches itself.

Figure 22 shows the components of the network. The general idea is that the network will receive information about the type of cell it is in (i.e., which sides of the cell have doors and which have walls), make a decision as to the direction of motion out of the cell (to simplify matters, I use compass directions instead of an "egocentric" orientation), and then predict the kind of cell it will be moving into. This prediction is then compared

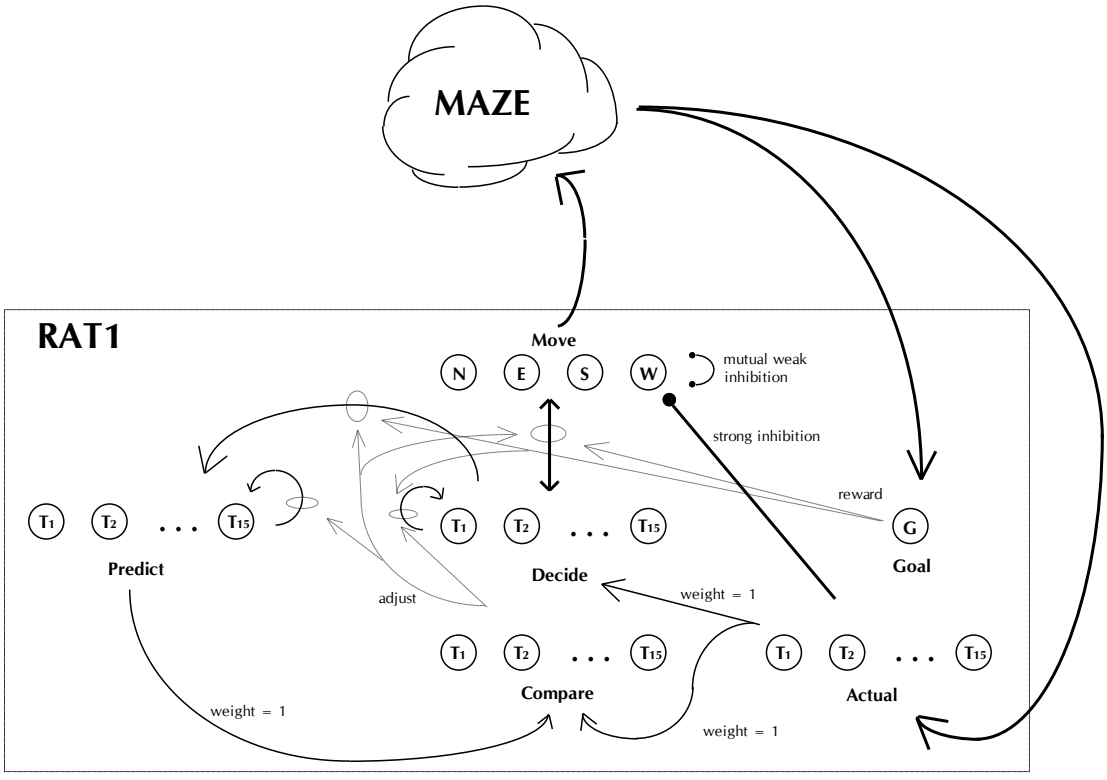
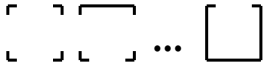


Figure 22. A proposal for a maze-learning net.

T_1, T_2, ..., T_{15} are 15 maze cell types: 

Broken lines are weight adjustment lines.

1. If predicted cell type = actual cell type, certain weights are adjusted away from zero.

2. If predicted cell type ≠ actual cell type, certain weights are adjusted toward zero.

3. If goal cell is reached, certain weights are rewarded (moved further from zero).

to the actual cell type moved into, and on the basis of that comparison, various of the net's weights will be affected. Also, if the goal cell is ever moved into, some of the net's weights are affected. The four move nodes constitute a "winner-take-all" subnet, activated by the "decision" subnet and strongly inhibited by various of the "actual cell type" nodes in order to ensure that a move chosen is a possible move. The move nodes are mutually inhibitory so that as one node's activation tends to grow, the others tend to diminish, with the result that eventually one and only one node will be active as the chosen direction.

The activity of the decision subnet is fed into the predictor subnet. Active nodes in this subnet are taken to represent the one (or more) cell types which the node expects to see after making the move indicated by the active node. They also act as a kind of "context" which is verified or falsified by the action of weight adjustments which take place after comparing the predictor's predictions with the actual new cell type. If this idea of "context" works, then the net will choose moves in a serial order based in some degree upon past modes and some degree on the present situation. That is, the net will "learn" a route from a given start cell to a given goal cell.

Once having learned a route, the start cell and/or the goal cell may be changed and the net allowed to investigate anew. The hope is that eventually the net will have a "feel" for the entire maze, and that it will be able roughly to orient itself anywhere in the maze after a number of moves.

An interesting feature about this proposed net is that it is given no knowledge about the size of the maze; the net, if it works at all, will work on different sized mazes. (There is obviously some limit to the maze complexity of course.)

I have been talking about the net as a proposal rather than as an accomplished machine, because at this stage it is only partially implemented. The basic structure of most of the nodes has been worked out, but it will be some time before all the pieces can be put together for some serious trials.

6. Maze Games

So far I have been talking of mazes as mazes — a series of paths from points to points. The series of paths may be simple or complex, but in any case the object is to move from some given point to some other. Although we may delight in the cleverness of a maze (perhaps it is embedded in a complex drawing, for example), as a puzzle or game it is rather restricted and can quickly become boring. If we wish to make use of the power of a computer in constructing and refereeing maze games, we ought to look to something other than the mere display of a maze which we are to traverse from entry to exit.

6.1 Blind Mazes

The first enhancement to computer maze games is straightforward: do not represent to the player his position relative to the rest of the maze — or else make this information at least partly hidden.

Such a maze game might, for example, display the perimeter of the maze and the position of the player inside the maze relative to the perimeter, but not display the individual cells within the maze, nor, of course, the paths from cell to cell. Such a maze we can call a **blind maze**. The player specifies the direction of desired movement from the present cell: up, right, down, or left (or else north, east, south or west). The player is moved accordingly if the present cell is open in the chosen direction. Otherwise, the player bumps up against a wall and remains in the present cell. (As an option, a wall which is bumped up against might then be displayed.) Allowing the player a maximum number of attempts to move in order to reach the exit cell would add an additional challenge. By adding a computer-simulated opponent (or by allowing for a second human player as an opponent), the game can be made even more challenging: one player tries to reach the exit cell (or goal cell) within a certain time, or within a certain number of moves; the opponent tries to catch the player (or get to the goal cell first). Both players move around the maze under constraints of darkness.

Instead of providing a "bird's eye view" of the maze, the player might be shown only the

"player's eye view". You will see only what is directly in front of you: either a wall, or else an opening to another cell. Only by turning around in your cell will you be able to learn where the cell's exits are located. In such a case, you might issue commands such as *rotate right*, *rotate left*, or *move forward*. Whether you are moving north, east, south or west will be known by the computer, which will keep track of your position and orientation, but can be known by you only if you keep track for yourself. The "player's eye view" might be enhanced by showing a perspective view of your position. Thus, if you are facing an exit from the cell, you will be able to see through into the next cell; if that cell has an opening on the facing wall, then you will be able to see through to the cell beyond that; and so on.

Adding an opponent (human or computer) who also roams the maze, will add challenge to such a game. In fact, such a multiplayer maze game would be a good candidate for a multi-computer game.

6.2 Wrap-Around Mazes

Certain very simple modifications to the traditional maze can be implemented. Consider, for example, the very simple maze in figure 23. One cell (cell 4) is established as the exit cell: moving north from that cell will constitute exiting the maze (and, presumably, winning the game). But in cell 9, movement is allowed only southwards. And in cell 10 there are walls to the east and west. Of course, there must be a wall on the east side of cell 10, because if there were not, then cell 10 would be the exit cell (or another exit cell). But wait! Why must moving east from cell 10 (assuming there were an opening there) really constitute an exit from the maze? Perhaps, instead, we can allow the maze to "wrap around" to the other side, such that moving east from cell 10 would put you into cell 6 (or 11). Allowing for such a possibility immediately transforms the very simple 5×5 maze into an effective 5×∞ maze. Allowing for wrap-around in the vertical direction as well will transform the maze into a ∞×∞ maze. If the player is given information only about the present cell (and, perhaps, about any cell which can be seen through an exit from the present cell), then even a simple maze will be very difficult to traverse, unless, by keeping careful records, the player begins to spot a repetition of patterns in the cells moved through.

Such an infinite wrap-around maze is extremely easy to implement on even a very small computer (I have put such maze games on programmable calculators), especially where the cells are internally represented as an ordered set of four walls, where, say, a "1" represents a wall and a "0" represents an opening. The computer need only keep track of which cell the player is in and the orientation of the player within the cell. Thus,

cell(i,1) = status of north side of cell i.
cell(i,2) = status of east side of cell i.
cell(i,3) = status of south side of cell i.
cell(i,4) = status of west side of cell i.

To move east from cell i, then, is allowed only if cell(i,2)=0, and a successful move then places the player in cell i+1.

6.3 One-Way Doors

Still another enhancement to a traditional maze is easy to implement, and this too can make even a small maze seem very complex, especially when a "player's eye view" is provided (instead of a "bird's eye view").

If each cell is represented by the status of its four walls (say, "1" for a wall, and "0" for an opening), then one expects that the neighboring cell will also have a corresponding "1" or "0". But suppose it doesn't. Suppose, for example, you are in a cell whose status indicates that the north side is an open door ("0"). Given the "player's eye view", and given that you are facing north, that means that you can look straight ahead into the

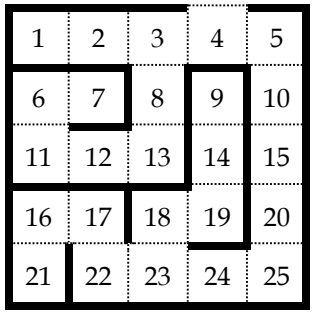


Figure 23. A simple 5×5 maze. Moving north from cell 4 will exit the maze, unless it is a wrap-around maze, in which case moving north from cell 4 would land you in cell 24.

next cell. Suppose you move ahead into that cell. But suppose that the new cell's status, instead of indicating a "0" for its south side (which, after all, would make sense, since you just came in that way), is actually "1". This means that if you were to turn around and face south, expecting to see through to the cell you had just come from, you would instead see a wall (or a closed door, or whatever it is that the game represents as a barrier). You have just come though a one-way door, and this makes traditional maze search algorithms far less useful, since, after moving through a one-way door into a cell never visited before, there will be no information about how to get back to the cell you came from without initiating a brand new search.

Even a very small maze which wraps around both horizontally and vertically, and which has a few one-way doors, can be an extremely difficult maze to solve.

6.4 Interpreted Mazes

A puzzle or game can be modeled on a maze without the game or puzzle seeming to be a maze. That is to say, the game will have a maze as its underlying structure, but the maze will not be apparent to a player of the game; the player will not think in terms of a maze at all.

Such a game can be constructed by *interpreting* (or translating) the constituents of a maze game into some other idiom. That is, being in a cell will be represented as being in some game situation with a number of options available (one for each open door). To accept a particular option is to move through the door into another cell — into another game situation with a (possibly new) set of options. (Or else each cell is a game, and entry to that cell from a neighbor is allowed only by successfully completing the game.)

For example, instead of geometrical motion through a maze, a game might be represented as a series of choices for trading goods. The player is given some object to begin with and then, for each open door in the cell, he is offered an object in trade. Accepting a trade which leads farther from the goal will result in some sort of net loss (say, in the dollar value of the item presently held), whereas accepting a trade which brings the player closer to the goal will result in a net gain. Backtracking might mean having to trade back for items previously held (presumably for a net loss).

Another example: A maze engine might underlie a bureaucracy game, where the goal is to contact Mr. X. At any given time (i.e., within any given cell) the player is allowed to contact only certain persons in the bureaucracy (various secretaries, undersecretaries, assistants, deputy managers, and so on). Some contacts lead immediately to dead-ends, i.e., closed doors in the cell. (Perhaps there are ten unhelpful persons in the bureau, and perhaps each of the closed doors in each cell is presented to the player as one of those ten chosen at random, so that very often the player is given a choice to contact, say, Assistant Deputy Under Secretary Smith, who turns out always to be most unhelpful. Eventually, the player will simply avoid bothering with Smith — which is to say that, in terms of the underlying maze engine, the player will have learned to recognize one kind of closed door.)

A maze engine might underlie instructional games. In order to gain entrance to a cell, a player has to successfully solve a certain kind of math problem. Easier problems lead farther from the goal, and harder problems lead closer to the goal.

Or each cell might represent a word spelling problem (or a word definition problem, or a language translation problem), where, as in the math game above, successfully completing easy problems takes the player farther from the goal, and successfully completing harder problems brings the player closer to the goal.

In general, each cell of the underlying maze engine might represent some activity or other (perhaps an entire game in itself, such as a Battle-the-Aliens game or a chess game, or…), and the game must be successfully completed in order to advance (i.e., move into the next cell), at which time another series of options is presented. And so on until the goal cell is reached. Probably multiply-connected mazes (i.e., mazes with more than one path from the start cell to the goal cell) would be appropriate for such maze engines.

Bibliography

Allen, S. and Allen, S. A., "Simple Maze Traversal Algorithms", *Byte*, Vol. 4, No. 6 (June, 1979), p. 36.

Even, Shimon, *Graph Algorithms* (Rockville, MD: Computer Science Press, 1979).

Jordon, M. I., "Serial Order: A Parallel Distributed Processing Approach", Technical Report 8604. La Jolla: University of California, San Diego, Institute for Cognitive Science, 1986.

Todd, Peter, "A Sequential Network Design for Musical Applications", in D. Touretsky, G. Hinton and T. Sejnowski, eds., *Proceedings of the 1988 Connectionist Models Summer School* (San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1989), pp. 76–84.