ON LETTING A COMPUTER HELP WITH THE WORK

Thomas C. Schelling

Teaching & Research Materials No. 12
Public Policy Program
John F. Kennedy School of Government
Harvard University

November 1972

Preface

If I had to teach about computers I would not begin with the "simulation model" that this paper is about, nor carry it to such length if I did. (But if somebody had to learn about computers, he would not choose me to learn it from.) The motivation of this paper is the other way around: it is written for people who already have acquainted themselves with the model--who have even, I hope, experimented with it by hand--and who, having invested that much already, would like to use it as a vehicle to which a computer might be harnessed, to see what computers can do and how they can be made to do it.

My advantage as teacher is that I so recently learned what I know and learned most of it trying to get computers to do the kind of job described here. To protect the reader from the grosser consequences of my amateur status I asked three people to look at this essay who had already helped me along the way. While I cannot cite them as warrantors, I can thank them. They are Alvin W. Drake, Associate Professor of Electrical Engineering, Massachusetts Institute of Technology; Tom Shemo, Jr., Manager of Computer Services for the John F. Kennedy School of Government; and James W. Vaupel, Instructor, Institute of Policy Sciences and Public Affairs, Duke University.

I have to add that one of the exquisite learning experiences of my life occurred one Sunday afternoon when, for three hours in a room with a blackboard, Jim Vaupel completely disassembled my "model" into its smallest components and reassembled it before my eyes as a set of instructions that a computer could follow. (He sailed for Europe next day and I was on my own.)

The paper that follows explains in rather minute detail, for people who have no idea of how a computer might be made to do a complex job, how a moderately complicated task can be made into a "program" that a computer can follow. Two cautionary points need emphasis.

One is that the machine is being used to help with the work but not to design the experiment or to draw the conclusions. The fact that the computer can produce results without error does not mean that it produces results with any significance. The machine can save a lot of time and trouble (and sometimes take a lot of time and make a lot of trouble); but the fact that the results come out of a computer does not give them any special authority--possibly the contrary, on account of the second point.

That is that the computer does most of its work away from our scrutiny. It processes raw materials and gives back a refined product. We lose some touch with the intermediate steps. We don't "see" what is going on, and may bury the most revealing part of the work in a magnetic memory and never know what is happening.

Let me illustrate. We could experiment by hiring a thousand drivers to evacuate a parking lot with alternative systems of exits and traffic lights; we might more cheaply simulate the process on a computer if we could specify every driver's reaction to his location and the traffic surrounding him. With the computer we could do multiple runs cheaply, varying the traffic controls, the number of vehicles, and the driving conditions. The computer could tell us things no driver would ever notice. But real drivers might notice things--important things--that the computer would never notice. The computer economizes by suppressing information. Unless we can guess what information will be most revealing, and ask for it, crucial events and processes will be unrevealed.

The experiment described in this paper is one that can be done by hand, at least in simplified fashion on a small scale. In this experiment there is no substitute for doing it by hand. Far more insight into the processes at work is obtained by a few hours of tabletop work than by the most elaborate project delegated to a computer. If, for this particular experiment, one had to choose between exclusive reliance on a computer and no computer at all, I would unhesitatingly recommend no computer at all. But if one had a hundred hours to devote to the experiment and a computer to help, I would recommend a few hours at the tabletop and ninety-odd hours in command of a computer.

There is a third cautionary point for anyone who may be tempted to let a computer help with the work. A computer program for a "simulation experiment" has to be elementary and exhaustive. An entire process must be decomposed into simple steps that are exactly specified. Often the program is designed by someone who is not the experimenter, in a "language" that the experimenter is poor at reading. The programmer has to make a myriad of small decisions, some rather arbitrary, in operationalizing the experiment for the computer. It is remarkable how easy it is for the experimenter and the programmer to have different ideas, and not to realize that they have different ideas, about details of the model. Perplexing results can occur because of some minor discrepancy between what the experimenter thought was happening and what the programmer made the computer do, each unaware perhaps that there was an alternative interpretation of the words they used to describe and to discuss the model and the program. Sometimes the small discrepancies have large effects; if so they may do more harm if undetected but may be less likely to go undetected. Sometimes "interesting" results were innocently planted by the person who designed the program. The three ways to minimize these difficulties are to familiarize the programmer with the purpose and design of the entire experiment, so that he acquires some personal judgment of what matters; to familiarize the experimenter, line by line, with the written program, whether he understands the program language or not; and to keep a lively awareness that, notwithstanding those efforts, the problem is always there. (As the paper will occasionally illustrate, the problem does not entirely disappear when experimenter and programmer are combined in the same person.)

ON LETTING A COMPUTER HELP WITH THE WORK

This is a guided tour through a computer program--a particular program, one that I constructed to have a computer do a job that I have done repeatedly without a computer. (I invite the reader to take half an hour to do it without a computer so that he will know what the job is that the program instructs the computer to do.)*

The computer can do it faster and more reliably, and it can keep records and compute statistics and print results faster and more reliably. It may take a hundred hours to develop the instructions with which the computer does the job; we use a computer only if we want to do the job so many times, or to do it on such a large scale, that a task that takes fifteen or twenty minutes is more quickly done by a machine demanding a minimum investment of perhaps a hundred hours. If we do it five or six hundred times we break even; if we do it five or six thousand times we're way ahead. Alternatively, we can redesign the task so that it is more complicated, taking several hours rather than fifteen minutes, and we can probably construct the program for the bigger job in the same time as it would take us to program the smaller one. The computer may do the bigger job so fast that we hardly notice the difference in computer time.

*It is described in T. C. Schelling, "On the Ecology of Micromotives," The Public Interest, Fall 1971, pp. 82-89, and in more detail in T. C. Schelling, "Dynamic Models of Segregation," Journal of Mathematical Sociology, 1 (1971), pp. 143-186.

An important way to make the job larger is to keep better records and to calculate better statistics. The computer can multiply and divide large numbers as easily as small ones; and because of its speed and reliability it allows us not only to do larger "simulations" but to keep more elaborate records.

When I say the computer is more "reliable" I mean that it almost never makes a mistake in following our instructions. But be careful: any mistake in the instructions we give it, as long as the instructions are logically capable of being followed, the computer will carry out exactly. It will never tell us that we gave it a stupid instruction. (With skill we can sometimes instruct it to recognize preposterous results and to send us a warning signal; but then it won't tell us if there's a silly error in that part of the instruction.) The computer is able to make silly mistakes at fantastic speed and with complete reliability, if somewhere along the way we instructed it to make those mistakes.

The computer furthermore performs only simple tasks. It can do an exceedingly complicated job, but only if the job is broken down into simple tasks. As an example imagine a maze. All passages in the maze are straight lines; all angles are right angles; all passages are of equal width; all passages that are blocked end in a wall perpendicular to the sides. You have to find a way out. You have searched the maze for some pattern, and you've concluded that there is no pattern you'll ever find. A computer can help you. The computer does not substitute a superior intelligence; it solves the problem the way a reliable, high-speed idiot would solve it.

The idiot would pick a route at random and follow it until he came to a blank wall. Then he would go back until he found another route and try that one, and continue this process until he came out at the end. Eventually, if he chooses routes at random, even if he repeats himself, the probability that he will find his way out approaches 100% if he keeps at it long enough. Of course, it may take him more than a lifetime. Since the computer can do in a few minutes what the idiot could do in a lifetime, a maze that is "humanly impossible" to get out of, except by sheer dumb luck, can be explored by a computer that knows how to follow the diagram. If the diagram can be expressed numerically, the computer can do it all with numbers. (The reason I used straight lines and right angles was so that we can visualize the maze on lined paper, with a number in every square, and can describe every position by its coordinates and can locate every wall on the side of a numbered square.)

The computer can do better than that; it can keep track of where it has been. This is what the idiot could do if he dropped sunflower seeds behind him and never went down a passage where the seeds showed he had already been. In that case, with a guarantee against going anywhere twice, there is an upper limit on the time the job will take. It may still be several lifetimes for an idiot, but a quick job for a computer.

An impressive thing about a computer is that it can do this kind of job so quickly.

An equally impressive thing about the computer is that a properly instructed idiot could usually do its job. (He'd just never get the job done.)

I don't mean an ordinary idiot, I mean a "super-idiot." I mean an idiot with the characteristics that (1) he can perform a rudimentary task if he is told exactly what task to perform; (2) he can keep his place in an orderly list of instructions, doing the next task after he completes each one; (3) he will stop, and perhaps ring a bell, if the instruction is incomprehensible in his language or if the task he is told to do is unperformable; (4) he never makes a mistake or gets tired.

His reliability derives from two characteristics: he never makes a mistake in doing exactly what we tell him to do; and he never thinks for himself.

* * * * *

What, now, are the rudimentary tasks that our computer can perform for us? To begin with, it can add and subtract, multiply and divide. It can keep in its memory the sums and products it obtains in order to perform repeated operations; it can recall the sums and products when it needs them. It can commit to memory the formulae for repeated arithmetical operations. It can erase things from its memory. And it can signal the keys of a typewriter to type out the answers we want.

Our speedy, reliable idiot can also keep running totals. That is, he can repeatedly add numbers and keep track of the latest subtotal. As a special case, he can count. By adding one to the total every time he performs an operation, he "counts" the number of times he performs it. He can keep track of "variables" that have names or labels, things like "X," "Y," "hits," "times at bat," and "batting average." He can assign numbers to these named variables; he can change those numbers at our instruction, substituting a new value for an old one, either arbitrarily or by adding something or subtracting something or multiplying when instructed. We can give him the historical number of hits and times at bat at the beginning of a game, and have him count hits and times at bat during the game, and at any time during the game, and at the end of it he can provide an up-to-date batting average for each member of the team and for the team as a whole. We can do this with a real game, or do it experimentally if we want to see how much difference it makes to an end-of-season batting average for a player to get one more hit during a season in which he plays a hundred games and comes to bat an average of four times per game. With a little ingenuity we can instruct the computer to signal a teletypewriter to type the information in a table with players' names spelled correctly and the numbers rounded to three digits.

The computer can compare two numbers to see whether one is greater than the other and, if so, which one. (This is equivalent to seeing whether a number is greater than zero, because we can always subtract one from the other and compare the difference with zero.) We can combine these abilities so that the computer will compare an old number with a new one, replacing the old one with the new one if the new one is larger. If each number has a name, the computer can substitute the name of the new number for the old name if the new number is larger than the old number, keeping track of "who" is the largest number that he has come across.

The computer can find a numbered entry in a list. We can have it compare the batting average of the seventh name in a list of names with the eleventh, pick the larger, and tell us the name that goes with it. In particular, it can start with the first name on the list and the number that goes with it, compare that with the second, "remember" the larger of the two and compare it with the third, and keep this up through the list, having one name and one batting average when it gets to the end, namely, the highest batting average and the name that goes with it. (We have to instruct it what to do in case of ties.) It can simultaneously keep a running total of hits and at-bats, and calculate a team average.

It can also keep a list of lists. We can number the teams in the National League, and it can pick the fifth team and find the player with the highest batting average on that team. In particular, it can go through the list of lists and, by a combination of the things I have described, find the team with the highest batting average, as well as the player with the highest average or the batting average of the whole League. And of course it can reject people, if we so instruct it, who have been at bat fewer than a hundred times. If we asterisk the players who are rookies, who are black, or who play first base, it can treat the asterisked players as a sub-list and find the rookie with the highest batting average, compare the batting average for black players with that for the League as a whole, or pick the team position that on the average has the highest batting average in the League.

What I have described is not everything that the computer can do but is typical of what the computer does. Notice how rudimentary it is. It can count, pick out a numbered item in a list, go through a list, keep running totals, perform some arithmetic, and that is about all. With ingenuity we can reduce a complex task to a series of these rudimentary operations, operations that will have to be performed repeatedly but with repetition that is fast and cheap.

One more thing the computer can do that is equally simple but of a slightly different character. In the same rudimentary way, it can perform operations <u>on its own instructions</u>. And at various points in its instructions we can provide it two or more <u>alternative sets of subsequent instructions and have it choose which instructions to follow</u> according to which of several variables has the largest number associated with it, or according to whether some number is positive or negative, or according to whether it has repeated some operation a stipulated number of times.

The reason we often want the computer to operate on its own instructions is that this permits us to economize on instructions. Let me show you how. I somewhat exaggerated the sophistication of this idiot. He may not know how to work his way through a list of 25 baseball players. He may just know how to pick a player who occupies a certain number on a list, like the 9th or the 25th. To make him work his way through the list, we tell him to find Player No. X and perform an operation. In advance, we gave X the value "1." So when told to go to Player No. X, he goes to

No. 1 and does whatever he is supposed to do. Now we want him to go to Player No. 2, and then No. 3. We tell him to replace the value of X with X+1. Since X was equal to 1, X+1 becomes 2. We then say, "Now go back and perform that operation again for Player No. X." Since we told him to change X to X+1, he changed X from 1 to 2, and now he does it for Player No. 2. Again he comes to the instruction, "Replace X with X+1"; he replaces 2 with 3 and comes again to "Do it again." If we had to say this 25 times for 25 players, that business with the X and the X+1 is a circuitous way of saying 2, 3, 4, 5. . .without using numbers other than 1; but we can economize.

Suppose he has a list of operations to perform in the sequence in which they occur on his list, and the 99th is to perform that operation on Item No. X in the list. He does it, whatever it is--it may be computing a batting average--and then he goes to Instruction No. 100. Instruction No. 100 tells him to change X to X+1. Instruction No. 101 says, "If X is no larger than 25, go back to Instruction No. 99 and take it again from there." Since there are 25 players on the list, as long as there are more players on the list he goes back and does the next one. If, when he adds 1 to the earlier number, he gets 26, he's finished the list; since he goes back to 99 only if he hasn't finished the list, he goes on instead to Instruction No. 102. This tells him to do whatever comes next in the program after he has performed that operation successively for each of the 25 players.

It only took three lines of instruction to have him do it 25 times. Of course, if we could just say, "Do it for everybody on the list, one after another in the order in which they appear," that would be even simpler for us. But he can be too dumb to understand that instruction, able only to do it for one player identified by his numerical position on the list, yet as long as he can add 1 to a player's number and can check whether the number he's at is no larger than the number of names on the list, he can perform two operations 25 times and cover the list just as well as if he knew what the words meant when we said to do it "successively for everybody on the list."

It may not strike you that the one instruction is more sophisticated than the other. But if the idiot already knows how to add 1 to some number he has in his memory, we don't have to add words like "every" and "successively" to his vocabulary to get the effect. It may take us a little longer to write the instruction, but it means we can get along with a much less articulate servant.*

* * * * *

Now, after all this introduction, let's go to the task we want performed. It is the one in which individuals are located

*Actually, to add "words" to his vocabulary, we'd put some expression like "for one after another" in his code book; and when he came across those exact words in a program involving 25 items he'd look up the recipe and find a short "subroutine" saying, "(1) Put X=1; (2) Do it--whatever he's to do 'for one after another'; (3) Change X to X+1; (4) If X less than 25 go back to #1; (5) Otherwise carry on." If the computer's standard language doesn't contain an expression for some regular operation that we want performed often in a program, we can usually define some terms of our own by writing short programs that will be triggered by some code words (somewhat as we might footnote, once for all, the exact definition of some term we wanted to introduce and use repeatedly, italicized to recall the footnote, in a set of written instructions to a helper).

on squares in a checkerboard pattern. We have a rectangular board divided into one-inch squares. It is n squares wide and m squares high, with a total of m x n squares. Some squares are blank, others have individuals located on them. Each individual belongs to some group, and the groups are numbered 1, 2, 3. . . . Maybe there are only two groups, maybe there are a dozen, maybe there is only one. Each group is identified by a group number. If a member of Group No. 3 occupies the cell three columns in from the left and two rows down from the top, we identify the number 3 with Row 2, Column 3.

Each individual is assumed to care about the group composition of the individuals who are located near him. Specifically, a member of Group 3 cares how many of his "neighbors" are also Group 3. For the moment suppose that, if they are members of other groups, he doesn't care which other group a neighbor belongs to. A member of Group 3 distinguishes among neighbors who are Group 3, neighbors who are not Group 3, and possibly blank cells in his neighborhood. Later we can change this so that a member of Group 3 considers a member of Group 4 different from a member of Group 5, perhaps considering Group 4 to be more like Group 3. That will be an easy adjustment to make, after we've learned to handle the case in which members of each group merely distinguish between "like" and "unlike" neighbors.

I shall assume that you have read something that explains the purpose of all this.* (I even hope you have done it by putting nickels and dimes on a sheet of paper consisting of one-inch

*The citations are in the footnote, page 1.

squares in a rectangle.) So I shall describe only <u>what</u> we are doing, not <u>why</u> we want to do it. The "why" relates to the research; the "what" relates to how we get the computer to help us do it.

What we do is to have every individual consider his own neighborhood, decide whether he's satisfied with the mix of like and unlike neighbors and, if he's not, to consider moving. To consider moving, he looks around for blank spaces that he likes better than the position he is in. If he finds one nearby he moves to it. Or, perhaps, if he finds one anywhere he moves to it. Which one does he move to? Maybe he moves to the nearest one that is decently satisfactory. Maybe he moves to the best space on the whole board, regardless of how close it is. Since others move, too, he can't be sure that the spot he moves to will remain satisfactory once he gets there, since he may lose neighbors that he likes or gain neighbors that he dislikes. So he may have to move again. Or maybe there's a spot that he likes but somebody beats him to it; or there's a spot that he likes but neighbors like himself move away before he can get there, or people unlike himself move into the neighborhood before he gets there, so he doesn't move there after all.

Members of each group have their own feelings about the kinds of neighbors they want. One group may have strict requirements for neighbors like themselves, another group may have modest requirements. The "requirements" may be stated in terms of the percentage of nearby neighbors that are members of one's own group.

They all keep moving in turn until everybody's satisfied, or until everybody who is unsatisfied can't find a place to move to. Maybe we drop the notion of "satisfied" and simply say that everybody compares every available vacant spot with the spot he's at, and moves to the best spot available if it's better than where he's at. Maybe we impose moving costs, expressed in a "currency" that is translatable into the "attractiveness" of different spots; and the moving cost is subtracted from the attractiveness in order to pick the spot that is best, considering both neighborhood and travel distance. And so forth.

Now this expresses the general idea. We can put nickels and dimes and pennies on the squares, letting the different coins represent the different groups. A "penny" counts the pennies in its own neighborhood, counts the nickels and dimes in its own neighborhood, calculates the percentage of the "local population" that is pennies, and decides whether or not it likes that percentage. It stays for the time being if it likes it--somebody may move in, or move away, and change his mind later--and if he does not like his neighborhood he searches the board, or some portion of the board near him, to see if there's a place he'd like to move to. If there is, he moves, if nobody beats him to it or if, in our program, nobody else can move until this one has had his turn.

Very simple. But if we ask an assistant to start moving the nickels and dimes and pennies in accordance with these rules, he is likely to tell us that our instructions are inadequate. Where does he start? Does he do all the pennies first and then the nickels? Does he start at the left of the top row and go across the top row and then do the next row, and so on down to the bottom of the board, and go back to the top row to start again? Does he look at everybody's position and pick the most dissatisfied, and move him, and then look for the one who is now most dissatisfied? And what do we mean by "neighborhood"? Every square is surrounded by 8 squares immediately; there is a 5 x 5 square surrounding that, containing 24 in addition to the individual's own spot. There is a 7 x 7 square that is still larger, containing 49 spots, one's own spot plus 48. One can "round" the corners a little, getting more nearly circular neighborhoods. What is the "neighborhood" that an individual cares about? Does he care about the "immediate" neighborhood of 8 neighbors in the 3 x 3 square more than he cares about the next 16 people in the outer part of the 5 x 5 neighborhood? Will we please specify precisely what ratio of like to unlike neighbors makes an individual satisfied? And, since we shall be dealing with integers --numbers up to 8 in the 3 x 3 square--is it simply a matter of percentage, or does it depend a little on the total number of neighbors an individual has? For example, if an individual is surrounded by blank spaces and has no neighbors, do we consider him completely satisfied, completely unsatisfied, or what? And so forth.

Finally--a very crucial part of the instruction--if our young employee keeps this up, scanning the board and moving the dissatisfied, when does he quit?

Evidently, if he scans the whole board and finds nobody who wants to move, scanning the board again will lead to the same result and he may as well stop. Suppose every time he scans the board somebody moves; then he has to go back and re-scan the whole board from the point of view of every individual, because everybody who moves can affect some other people, who may have been satisfied a moment ago but are no longer satisfied. Does he quit at 5:00? Does he stop at 5:00 and come back tomorrow and spend another eight-hour day? If he finds that each time around he reverses the move he made the last time around and is merely repeating a cyclical process, and he sees that, no matter how long he goes on, people will be moving in the same circles, can he simply report that he has reached an endless cyclical process, one that is predictable from here to eternity, and stop? It would be a shame to go away on vacation and come back and find that he had been working an eight-hour shift getting nowhere merely because we didn't give him instructions to quit.

So the first thing we have to do is to specify all of the "parameters" of the problem exactly. We have to give the length and width of the board, measured in number of squares. We have to state how many different groups we want, how many members of each group and how many blank squares. For each group we have to state its exact conditions for being content where it is or for moving; and, if it moves, the exact portion of the board (perhaps the whole board) that it will scan for a place to move to, and the exact way that it evaluates every spot, both in terms of neighborhood and in terms of moving distance if moving distance matters. We have to state the precise order in which we do this for different individuals; we can run through the board systematically, or we can do it for one whole group before going on to the next group, or we can do it by picking members at random or according to some specified plan. But since our employee isn't paid to "think," but only to do what we tell him, we have to give him absolutely comprehensive instructions to cover every contingency, so that he never is at a loss what to do next and never has to do his own thinking.

We need some pattern to begin with. Either we must put the nickels and the pennies and the dimes on the board, or we must tell our "employee" how to put them there. Maybe we want our computer to randomize the pattern; maybe we want it to locate them in some regular pattern, and can instruct the computer exactly how to lay down nickels and dimes and pennies, i.e., to assign numbers like 1, 2, 3 to the different successive cells in the rows and columns. Or maybe we tell the computer the group number of the occupant in every cell, each cell identified by row and column, with the number "0" being used to denote a blank cell.

Now what does the computer do?

It does the same thing--something--over and over again for each individual on the board, in some order or sequence. We have to tell it the exact order in which it performs the operation for every individual. It can even be told to choose them at random, if we can arrange for it to find a randomizing process. But it will do this "something" one at a time for an individual. (More complicated: it might study the situation for every individual, and perform the action for the particular individual whose gain in moving were calculated to be greatest. It ought to be clear that we can make it do that, if we can make it take them in a particular order, so let's stick with a particular order.) So we have told the computer how to pick the "next individual" for which it does the basic operation: now how do we describe that <u>basic operation</u>?

* * * * *

We have, let us say, the individual located at Position 3,5. This means the cell in the third row, fifth column. Every square on the board has three numbers associated with it, numbers that will identify it and the group to which its occupant belongs. One is the number of its row, counting from the top; another is the number of its column, counting from the left. The third is the group number. We have a computer that can count rows and columns, so when we say, "Individual 3,5," it counts down the third row from the top and the fifth column in from the left, and identifies the group number of the individual who is located there. (It actually doesn't use a board for the purpose; it simply has something like a table stored away in its memory. The table has rows and columns. Essentially, each row is a "list" of individuals arranged in numerical order. The "row number" tells the computer which <u>list</u> to examine; the "column number" tells the computer which numbered <u>individual</u> on that list to look for. Metaphorically, we ask it to find the group number of the individual in the third row and the fifth column; numerically, we ask it to find the fifth individual on the third list. <u>We</u> can think in rows and columns, in two-dimensional space. <u>It</u> can work in terms of card catalogs, code books, or any other kind of "lists.")

If we want it to start in the upper left corner and go across the top row, then come back from right to left in the second row, then go from left to right in the third row, and so on down the board, we tell it to start with "List No. 1," and take the individuals in order, then to progress to List No. 2 and take the individuals in reverse order, go on to List No. 3 and take them in direct order, and so forth until it runs out of "lists." We have already seen, above, how to arrange so that it knows when it's at the end of a list, or has finished the list of lists and should go back to the first list again.

So we have a way of telling it how to pick an individual. We have a way of formulating that instruction in some regular fashion, such as having it progress through the lists.

So he's found our "next" individual. What does he do for him? The first step is to find out whether or not he is satisfied, or how satisfied he is. Then, if he is not satisfied, we institute a search for someplace else to go. Then we move him. Then we move on to the next individual.

Step 1 is to evaluate the individual's present neighborhood. This is done by counting the number of individuals belonging to the same group in that neighborhood, counting the number of individuals of different groups, constructing the percentage that the first is of the total, and comparing that with some minimum percentage at which he is satisfied. How do we do this?

We have already defined what we mean by "neighborhood," and suppose that it is merely the 8 neighbors surrounding this particular individual in the 3 x 3 square. The computer has to look at each square and see whether or not it is occupied, and, if it is occupied, whether or not it is occupied by an individual like the individual for whom we are performing the process. If we were doing it ourselves, we could draw the boundary of the 3 x 3 square, and then run through the 3 squares in the first row, the 3 in the second row, and the 3 in the third row, keeping three running totals as we went (or perhaps just two totals, knowing that the blanks would always be equal to 8 minus the sum of the other two numbers, if we were interested in the number of blanks). To do this the computer has to know how to perform two operations. First, it must know how to <u>find</u> the 8 cells that we want to examine; second, it must know how to <u>count</u>. Consider the first problem, finding a cell to examine.

Suppose we're dealing with the individual in Row X, Column Y. Going through the 3 x 3 square surrounding him, and starting in the upper left corner of that little square, we want to look at the individual located at Row X-1, Column Y-1. Next we want to look at the individual in Row X-1, Column Y. Then Row X-1, Column Y+1. Then we go to Row X, Column Y-1; then we want to skip Row X, Column Y, because that's our own individual, and we don't want him to count himself as a neighbor. Then Row X, Column Y+1. Then down to Row X+1, going through Column Y-1, Column Y, and Column Y+1. Since our own individual is identified by the numbers, X and Y, it is easy to identify his neighbors by the numbers X and Y with a 1 added or subtracted here and there. This we know how to do. We tell the computer, for Individual X,Y, to find the square located at X-1,Y-1 and perform a counting operation. When he has counted the individual located in that square, he adds 1 to the second of the two numbers (he adds 1 to Y-1), and does it again; then he adds 1 to that second number again, getting Y+1. Then he must know that he's finished that row--that he has done all that he is to do when the first number is X-1--and add 1 to X-1, making it X, and go back to Y-1.

Column

				Y-1	Y	Y+1		
	*	*	*	*	*	*	*	*
X-1	*	*	*	*	*	*	*	*
X	*	*	*	*	0	*	*	*
X+1	*	*	*	*	*	*	*	*
	*	*	*	*	*	*	*	*
	*	*	*	*	*	*	*	*
	*	*	*	*	*	*	*	*
	*	*	*	*	*	*	*	*

We do this by starting with the numbers X and Y, and subtracting 1 from each. Specifically, we say, "Let P equal X-1. Let Q equal Y-1." Then we say, "Count whatever you are supposed to count on Square P,Q." That is, go to the square in Row P, Column Q, which is the square immediately to the upper left of X,Y, and see who's there; make a note of it. Next we say, "If Q is less than Y+1, increase Q by 1." Then, "Do that operation again." (Go back to Operation No. 66, or whatever it was, where we said, "Count and keep track of who's there.") At the beginning, Q was equal to Y-1; it is therefore less than Y+1; therefore the instruction says to increase it by 1. So Q is changed from Y-1 to Y.

And the "counting" operation is repeated. Whoever is located in Row P, Column Q, which is now Row X-1, Column Y, is counted. Is Q less than Y+1? Yes, since it is equal to Y. Therefore we add 1, changing Q to Y+1, and perform Operation 66 again, which is to count the person located at Square P,Q, which is Square X-1,Y+1.

We have now counted, by membership in the different groups, any individuals located in those three squares in the row just above the individual located at X,Y. The instruction says to increase Q again, if Q is less than Y+1. But Q is not less; it is equal to Y+1. So the machine does not increase Q. What does it do? It simply moves on to the next instruction, which says to increase P by 1. P, which was equal to X-1, becomes equal to X. We also instruct the computer to subtract 2 from Q, so that Q goes back to Q-1, and we "sweep" the three squares on Row X, starting with the square to the left of X,Y, and finishing with the square to the right of it.

We don't want to count the individual at Square X,Y; it's himself, not a "neighbor." We need a special instruction for that, which I'll come to in a minute. The same way that we increased Q by 1, until Q was equal to Y+1, we sweep this second row. When we get to the space to the right of the individual we're doing this for, the instruction puts Q back to Q-1 and adds 1 to P, which becomes P+1, and we do it on the third row of our 3 x 3 square.

We need a "skip" rule and a "stop" rule. We skip the square, X,Y, by an instruction that says, "If P = X and Q = Y, skip to that instruction where you add 1 to Q, passing over the 'count' operation." There are different ways to do this; if the computer doesn't understand the expression, "If blank and blank, then . . .," we may have to put it through a "screening" procedure, where it proceeds normally if P does not equal X, otherwise it looks to see whether Q equals Y, proceeds normally if it does not, and at that point skips if P and Q failed both tests.

For "stopping" the procedure, we do with P what we did with Q. We said, "If Q is less than Y+1, add 1 to Q," otherwise the next instruction said to put Q back to Q-1, and add 1 to P. We need an instruction that says, "When Q is equal to Q+1, if P is equal to P+1 you're done, do the arithmetic on the basis of what you have already counted." Thus every time it comes to where Q is equal to Q+1, it either goes back to the next row by increasing P, or, if P is already up to P+1, it skips the instruction about adding 1 to P and doing it again, and goes on to the next operation.

Before we look at <u>that</u> operation, let's look at how the computer "counts." Every square has an associated number, namely, the group number of the individual on that square, if there is anybody on that square. If there's nobody on the square, we can use the number 0. "Counting" involves this kind of instruction. Keep in mind that the computer can look at two numbers and tell whether or not they are equal, just as it can look at two numbers and tell which is larger. We "name" two variables for the computer, one is named "Like" and the other is named "Unlike," and we can abbreviate them L and U. For each individual whose neighborhood is to be "counted," we start with L = 0 and U = 0. This is where anybody starts when he wants to count. He adds 1 to his subtotal every time he finds something worth counting; and if his starting subtotal is 0, his ending subtotal is equal to the number of things he counted. So we put L and U at 0, and tell the computer, if the group number associated with a square is the same as the group number of Individual X,Y, to add 1 to L. If the group number is different from the group number of Individual X,Y, look and see whether or not it is also different from 0. If it is different from X,Y and different from 0, add 1 to U. (If it is equal to 0, it doesn't add anything to anything.) Then it goes on to the next square, in accordance with the procedure described above, adding 1 to Q, or adding 1 to P and going through the next row.

Since L started at 0, and increased by 1 every time an individual was encountered who had the same group number as Individual X,Y, when we've been through the 8 spaces surrounding our central individual, L has increased by 1 for every like individual encountered. Similarly for U. And subtracting L and U from 8 we have the number of blank spaces, if we want it.

Now we have counted, and we have our data. What do we do with them? We tell the computer to divide L by L+U. This gives us a number from 0 to 1, the fraction of total neighbors who are of the same group as Individual X,Y. We now bring in a piece of "data" that we put into the project, namely, the percentage of neighbors like himself that a member of the group represented by X,Y needs in order to be satisfied. Suppose it is .5. We ask the computer to compare .5 with the number it obtained when it divided L by the sum of L and U; if .5 is larger than that ratio, the individual is dissatisfied, and we invoke the procedure for dissatisfied individuals. We'll get to that in a minute. If .5 is less than the ratio obtained by our count, the individual is satisfied and we leave him and go on to the next.

Let's look at <u>how we go on to the next individual</u>, since we're going to have to do that anyway, even if we perform for this individual the search for a better place to go and move him there. Let's suppose, therefore, that this individual doesn't need to move--that's the same as supposing he has already found a place and moved--and see what we would do next.

You have probably guessed. We change Y to Y+1, leaving X as it was. This moves our reference numbers to the individual in the same row as the one we just worked with, one square to the right. And then we repeat the whole procedure.

We need one little safeguard. If the individual we just worked with was already in the right-hand column, there is nobody to his right. If there are a total of n columns, Y was equal to n. We need an instruction like, "If Y = n, change Y to 1, change X to X+1, and go back to Instruction No. 55." This means that it's come to the end of the row; it goes back to the first entry in the next row and does again what it's already done.

[This ought to remind us of a problem we skipped a minute ago. Because I pretended we were dealing with somebody in the third row and the fifth column, suggesting that there were eight surrounding squares, we didn't consider what happened if there were only five columns. In that case the individual was on the right-hand edge of the territory. The computer can't look in the square to the right of that individual, because there is no square. Nor can it look for an individual in the square one row up and one column to the right, because there is nobody there either. Indeed, if our individual were located in the upper right corner there would be nobody in Row X-1, because there is no Row X-1, nor would there be anybody in Column Y+1, because there is no Column Y+1. So we need a subsidiary instruction; after putting P equal to X-1, we have to say, "If P = 0 skip the next few operations and add 1 to P." Similarly, "If Q = n don't increase Q by 1 but put it back to Y-1 and increase P by 1 and start the next row." This is all straightforward but important, because if we forget it the computer will do one of two things, depending on the exact instruction that we gave it. It may stop the whole program and send us a "diagnostic message," telling us that it has an impossible instruction. That will be a nuisance, and we shall have to find out what's wrong with the program before we can get the computer to do the job. Worse--again depending on the exact language we used--the computer may find that we put nobody in any place labeled P,Q, where Q is greater than n, and proceed to treat it as a blank space, giving a 0 value to anybody who was never given any other number, and count a blank space. If blank spaces don't matter in our arithmetic, no harm is done; but if we want to calculate population density in the local neighborhood, we have treated territory beyond the boundary as though it were unoccupied territory within the boundary, and maybe we didn't mean to. And because the instruction is feasible, the computer never tells us that it's doing something we didn't intend. It is merely interpreting an instruction literally, in accordance with certain rules of interpretation--rules that govern contingencies, and that we may not have been aware of or alert to--and instead of stopping and ringing a bell because we told it to do something impossible, it does something possible that we didn't intend it to do.]*

* * * * *

Now, here is where we are. For one individual we've found out whether or not he's satisfied. If he's satisfied we've moved on. If he's not satisfied we have reached the guts of the program. We must search the board to see whether he'd like to move somewhere, and move him. Once we have done that we've done all there is, except for keeping records. If we can move this individual to the place he'd like to move to, we've done the only significant thing there is to do. We simply go on to the next person and do the same thing for him if necessary. Then on to the next person. And we just keep it up until nobody wants to move or until, by prearrangement, we call it quits.

We shall, of course, need a way to go back to the upper left corner when we reach the lower right corner; but we've had enough practice with that sort of thing so that it's clear that, when $X = m$ and $Y = n$, instead of adding 1 to Y, or switching back to the next row and adding 1 to X, we identify ourselves as in the

*The trouble may be even harder to diagnose if the overall format is no larger than the rectangular array we are working with; instead of getting a zero beyond the right margin, the machine may pick up the first entry in the line below.

lower right corner and put X back at 1, Y back at 1, and start a new inning of the same game.

So: we now need to search the board for blank spaces that are better than the space presently occupied by our individual, find the best among them if there is one, and move him there.

Suppose we had already found the best among some blank spaces. How would we move him? Of course, we don't physically "move" him. The computer has no hands. Rather we make him "disappear" where he was and "appear" where he wants to go. If a chess player hands his written move to a referee, it might say, "Move the knight from king-bishop 3 to king-rook 4." The referee might say, "I don't know the word 'move'." The chess player might then say, "(1) Let there appear a knight on king-rook 4. (2) Let there be nobody at king-bishop 3." If our individual belonging to Group 2, located at Row 3, Column 5, wants to move to Row 9, Column 10, we tell the computer to change the group number at 9,10, from 0 (which it must have been, if the square was blank) to the same group number as presently occupies 3,5, namely, 2. We then tell it to change the group number at 3,5, to 0. In effect, we have "moved" the individual. Henceforth the computer's record shows that an individual of Group 2 who was at 3,5 has disappeared from there, and an individual belonging to Group 2 has newly appeared at 9,10.

Now that we see how we <u>could</u> move him, let us see how to <u>find out where to move</u>. The general idea is that we look, on his behalf, at every blank space on the board. We evaluate it from <u>his</u> point of view; we count the neighbors like him he would have and the neighbors unlike him he would have at every available blank space. We keep a running record of the "best" blank space we have found. That is, we evaluate the first blank space; we do it for the second, compare them, and remember the location and the score of the better of the two; we then evaluate a third blank space, compare it with that one that we saved, and keep the better of the two. And so on for the fourth. To "save" the better of the two means that we remember the location and the value. We might do this by giving names to the three pertinent numbers. The location is recorded by row and column, and we give a variable named "R" the row number of the square we want to save, and we call "C" the column number of the square we want to save. The score we can call "S." If the latest square we examine is not as good as the last one we saved, we leave R, C and S alone. When we find a square that is "better" than the one we saved, we have the computer change R to the row number of this new and better square, change C to the column number of this new and better square, and change S to the score value of this new and better square. That "saves" this new square in the memory, and "forgets" whatever the square was that we were carrying along as the best so far.

(We need a rule to cover ties. We can be arbitrary; keep the old one, keep the new one, keep the nearest one to the individual on whose behalf we are searching the blank squares, or even pick a random number and take the new square if the number is odd and keep the old one if the number is even.)

When we get to the end of the board, having looked at every blank square, we compare the "best" available space with the space the individual is already at, and move him as described earlier if the best blank space is better than where he is. Otherwise we leave him alone. Alternatively, we can begin this process with the score of the individual's own position, letting R, C and S denote the individual's own row, column and surrounding score (using a slightly different scoring formula if we want to give the benefit of the doubt to staying put, or if we want to make the "improvement" exceed some minimum cost of transfer), and compare the first blank square with his home square, "saving" his home square until one is found that is better. If none is better, we complete the process, having "saved" his original site.

[Here is a good place to illustrate the kind of perplexing and infuriating error that can creep into a program. Suppose we devise the program the way I last described, namely, by evaluating sequentially the individual's own square and all blank squares, "saving" in the computer's memory the best spot found so far. At the end we "move" the individual. If he is to stay put, we might simply "move" him to his own square. But be careful! If we "move" him the way I described, the computer carries out two operations. First it "changes" the group number associated with the individual's own square to the group number of that same individual. "Change" is the wrong word, of course; it substitutes for the group number that was on that square the group number of the individual in question, which is exactly the number that was already there. It replaces 2 with 2 or 1 with 1 or 3 with 3. No harm so far; being an idiot, it doesn't mind carrying out foolish instructions, and the computer has no way of "noticing," or doesn't care, that we have asked it to replace a number with the same number. It is that second step in the process that can cause trouble. Now that we have caused the right group number to show up on the square that the individual "moves" to, namely his own square, the computer has to make the individual "disappear" at the square he started from, and it does this by changing the value on that square to 0. So, if our individual was a member of Group 2, the computer changes 2 into 2 and, immediately afterward, changes the 2 to 0. The net result is that the individual has "moved away." He is no longer on the board! The population of Group 2 has been reduced by 1. Individuals who move to another square are still with us; individuals who do not move at all, i.e., who want to "move" no place, get erased. Unless there is something in the dynamics of moving that causes everybody, after some point has been reached, to keep moving, the entire population will disappear. When the computer has finished the entire project, we ask it to type out, in a rectangular array of rows and columns, the group numbers corresponding to all the spaces, and we get a rectangle of 0's.]

The only part of the process we haven't covered is that of locating, one at a time, all of the blank squares. But by now that must look easy. We let the computer start in the upper left corner and go across the rows from left to right. At each square we have the computer test whether the group number is greater than 0; if it is it skips the evaluation process and goes on to the next square, because if the group number is greater than 0 the square is not blank and is not a feasible destination. When the group number is 0, the computer performs the kind of evaluation that we have already looked at, scrutinizing the 3 x 3 square.

[Here I can illustrate another pitfall. It probably wouldn't occur to you, as it didn't to me. Only when some strange result arouses your suspicion do you begin to look for what may have gone wrong. And very possibly you will find it only when you do as I did, following a bit of advice I found in a book on the subject, which was to go through one step of the program completely on paper--not doing it on paper the way one would do it on paper if he had no computer, but following on paper in exact detail the instructions I had given the computer. That way I found what I had done wrong.]

[What can happen is this. Suppose there is a blank square just to the right of the individual on whose behalf we are conducting the search. When we count the neighbors who surround that blank space, we count himself! In other words, if the blank space is a "neighboring" space, then <u>he</u>, where he is right now, is a <u>neighbor</u> of <u>that</u> blank space. If a real person enters a vacant house next door and looks around to see who his neighbors would be if he moved there, not being an idiot he will know that he can't count himself, because if he moves he won't be next door to himself. But the idiot won't recognize that the individual on whose behalf we are searching can never be his own neighbor. We only told him to count "people" in the houses adjacent to any given blank square. And if by coincidence it turns out that, not counting himself, the two squares are equally attractive --the one where he already resides and the one adjacent to it--and if he correctly omits himself as a neighbor in evaluating his own neighborhood but neglects to omit himself as a "neighbor" when he evaluates the neighborhood of a neighboring square, the blank square will always have one more neighbor like himself than the square he is at. Under the rules, he will move to it. Instantly the square he vacated acquires a "neighbor" like himself. It is himself. The space he moved to "lost" a neighbor and the space he moved from "gained" a neighbor. If the rest of his environment stays put, he will hop back and forth forever, or until so many moves have been made that the computer runs into its instruction to stop.]

[This is an easy error to correct, but it is an easy one to make and it may not be an easy one to find. It is the kind that you may find if, keeping your wits about you, you put yourself in the idiot's place and do his job laboriously.]

[I have mentioned a couple of these ways that the program may go astray, to emphasize how cautious you may have to be when your servant will literally perform every act in exact compliance with what you have told him to do.]

Now let's recapitulate. The guts of the program is a little procedure whereby the computer examines, one after another, the eight squares surrounding a given square. It computes a "score" as simple or as complicated as you please, depending on the group numbers of the individuals in those squares. This little process is then embedded in a surrounding process; the surrounding process is the search of the whole board for blank squares, performing that central process just described and keeping track of the "best" location. Finally we have some arithmetic that is equivalent to "moving," if moving is indicated.

Now we have taken care of an individual. We have a still larger process in which all of that is embedded, a process by which we do this for every individual on the board, in some specified order. Then that in turn is embedded in a still broader set of instructions that tells the machine when to keep going and do it all again, and when to stop.

Basically, it can be told to stop when everybody has had a chance to move and nobody did move (a very easy kind of instruction to provide), or after everybody has had some specified number of opportunities, say a dozen complete "innings" in which everybody had his chance to move (also an easy kind of instruction to give, the computer just adding one to some counting variable every time it switches back to the upper left corner, with a stop rule when that counting variable hits the limit), or (perhaps too complicated for us to write out, but no problem for the computer if we can write it out) when some endless repetitive pattern of movement has been reached that looks as though it will go on forever.

All of this can be further embedded in a packaged research program. It is easy (if tedious) to tell the computer to use successively small, medium and large rectangles, or to use two, three or four different groups successively, or to use two groups of equal, slightly unequal and drastically unequal size, and to use values of .3, .5 and .8 successively as the "satisfactory" fraction of neighbors like oneself, for the two groups separately. This is just a matter of the computer's storing its results and changing some basic "data" at the end of an embedded program, and going back to the beginning and doing it all again. And, as we could within the program itself, we can give contingent instructions in our research program, e.g., enlarge the rectangle successively by two rows and two columns as long as each enlargement causes a significant change (precisely defined numerically) in some specified average values, stopping when further enlargement "appears" to make no further difference in the results. And so forth. And again it can calculate averages, standard deviations, correlation coefficients and more complicated statistics, printing out what we want and even plotting diagrams. (Diagrams are plotted by nothing more subtle than repeated use of the teletypewriter space bar: a value of 14 is denoted by typing 13 spaces followed by a dot or an asterisk, or 14 hyphens if a bar diagram is desired.)

* * * * *

And that's it. That's the "computing." But where are our results? If it were a real idiot pushing nickels and dimes around on a checkerboard, we could walk into the room and look at the checkerboard, copying or photographing it. And if he had kept an account of the moves, we could look at his records of how many people moved, how much total distance was moved, how many neighbors like themselves people had at the beginning and how many they have at the end, and all of that. But since this was a computer, all of our data exist in some magnetic form at the other end of a telephone line.

The computer has our results; now we have to instruct the computer, like the dog that has retrieved our partridge, to drop it in our hand. We have to instruct it to type out the results that we want. (We have to make sure that it types out only what we want, because it may have a great quantity of useless individual statistics in its memory, of which we want only some totals and some averages.)

At this point it should be clear that we could have kept running totals of all the things we added and all the things we counted, and if we know the formula for an arithmetic mean, or of a correlation coefficient or a standard deviation, we can instruct the machine to multiply certain things and add certain things and divide by certain things and give us the resulting statistics, rounded off to whatever accuracy we like.

We can also instruct the machine to "label" the numbers it prints out. We can do this by having the instruction, "Print the average number of moves per member of Group 2," preceded by an instruction like, "Print 'Group 2'; print colon; print space." And a little earlier it would have printed a table heading with an instruction like, "Print 'Average Number of Moves'." Of course we wouldn't actually say, "Print the average number of moves for Group 2." We would say something like, "Print m(2)/n(2)," where n(2) is the number of individuals belonging to Group 2, and m(2) is the "counting variable" that kept track of another move every time a member of Group 2 made one, and represents the total moves made by that group. The computer has been given a language so that, when two variables are separated by a slash, the first is to be divided by the second; and so that, if that expression follows some recognized verb like "print" or "type," it goes to a list of "m( )" values and finds the item "m(2)," gets n(2) from a list of "n( )" values, divides the former by the latter and activates the teletypewriter keys to type out the result. And there are techniques by which to instruct it to print only to the third decimal place, or to print hyphens if no such number exists, just as we can arrange to have it space things across the page and down the page in the form of a neat table with headings.

If we want the computer to give us a picture of our "town" with the individuals belonging to different groups distributed around the town, we need a way to convert lists of numbers, and lists of lists of numbers, into a visual pattern. If the pattern is two-dimensional, like a checkerboard, and if the lists are of equal length, it is easy to "list" every individual in rows and columns and interpret the rows and columns as spatial coordinates. Every individual's "location" on a list is denoted in the computer's memory by two numbers, one the number of the "list" he is on, the second his position number on the list; if his numbers are 3,5, he is in the fifth position on the third list. We don't want his name--he has no name unless we put a name in our program, which we did not--all he has is a number identifying the group that he belongs to. So if the group number of the individual in the fifth position on the third list is 2, the computer types the number 2 as the fifth entry from the left in the third row from the top. Because a teletypewriter goes across the page from left to right and then switches down a line, we can instruct the computer to run through List No. 1, typing successively the group numbers of the individuals it encounters on that list in the order in which they appear, with a space or two between them on the page for visual convenience, and to switch to List No. 2 when it comes to the end of List No. 1, returning the teletypewriter carriage and dropping down one line. It then types, as the second row of our matrix, the group numbers of everybody on List No. 2, that is, of everybody whose first coordinate is the number 2.

It is worth remembering here that there is nothing spatial about the material the computer worked with. It worked with what I have called "lists" of numerical values. It could equally well print out, separately for each group, a list of the coordinates of the individuals comprising that group. In the computer's memory, Row 2 is not necessarily "next to" Row 3; Row 3 comes next in a search process or a typing process only if, for convenience of programming, we have the computer "count by 1's" in order to guide itself exhaustively through the set of lists. And if we had worked with a three-dimensional space, letting the "individuals" occupy rooms in a building, there would have been no way to type out on a flat sheet of paper a good visual representation of everybody's location. If we visualize the building as having express elevators, Floor 22 may not be closer to Floor 20 than Floor 30 is. And so forth.

* * * * *

That pretty well describes how we convert what might have looked like a fairly complicated procedure to a series of small instructions that an idiot might carry out, the operations being things like counting, comparing numbers, substituting one number for another, and skipping or acting out certain instructions according to whether or not some number is 0 or has reached some upper limit. A very important kind of instruction is, "Go back and do it again for the next one," which is typically accomplished by adding 1 to some number, or putting some number back to 0, and cycling the computer back to an earlier point in the instruction sheet. In actual practice, the instructions all have to be written in a "language" that itself has already been "programmed" into the computer. This is essentially a "coding" operation. Procedures that are complicated and that would be tedious for us to write out every time we wanted them are given a code name, usually one that consists of short English words and punctuation marks; we memorize the code name and let the computer use the code book to find the recipe for the task we have in mind.

* * * * *

Students who sit at a computer console for the first time often cannot dispel the impression that the computer is talking to them, asking them questions or accusing them of error. A program like the one I have described might begin, once a "run" has been properly initiated, by typing out some explanations, telling what the program does and what the researcher at the console is going to have to do. It may then ask a series of questions, such as, "How many rows and columns?--Type two numbers separated by commas." The student types two numbers separated by a semicolon and the computer comes back with the charge, "Illegal formula, try again." The student does it right this time, asking for a long, thin array consisting of 5 rows and 40 columns. The computer comes back and says, "Only 20 columns fit on a page. Not programmed to break town in two and print right-hand half below left-hand half. Can do the job and get the statistics, but cannot type it out for you. Alternatively, if it makes no difference to you, use 40 rows and 5 columns; that can readily be typed." The language may be cryptic or eloquent, in the style of telegraphs or of essays, and it may take an infuriating length of time for somebody who, at the first few words, realizes what he has done wrong but has to sit and watch the whole message be typed out.

The computer may sound even more human. Suppose in a 10 x 10 array you want four different groups and some blank spaces, and when the computer queries you as to how many individuals you want in each group, you inadvertently give numbers that add up to 100. The computer might come back and say, "Can't you count? You haven't left any blank spaces. Do you want me to search 100 squares for blank spaces, for 100 individuals, making 10,000 stops along the way, when any fool can see that I can't ever find a blank space for anybody the way you've set it up?" It sounds human until you look at the written version of the program. There you will find, perfectly deadpan, an instruction written by the programmer. The instruction goes something like this. "Let n equal n(1) plus n(2) plus n(3) plus n(4). If n less than [rows] times [columns] proceed. If n greater than or equal to [rows] times [columns] print '. . .message. . .'." And the message is the one I wrote above.

In other words, the computer is simply copying the message that it's to transmit if some number equals or exceeds another number. It will reliably misspell any word that was misspelled in the program, just as it will misspell your name if somebody got it wrong the first time on the addressograph. So keep in mind that, whenever the computer seems to be "talking" to you, it is simply copying pertinent messages that were written into the program as messages to be typed out if certain conditions arise. If some program were designed, let us say, to help you calculate the relative merits of two retirement systems, it might ask you to type your birth date into its memory at some point along the way. A whimsical programmer could have arranged for the computer to type out "Happy Birthday" if it's your birthday. But nobody's wishing you happy birthday. If two numbers match, namely your birth date and the date of the day you're sitting at the console, there is a "canned" signal to the teletypewriter keys. To reassure you that nobody is eavesdropping, the programmer might have had the computer say, "This is a recording. The message says 'Happy Birthday.' This recording is automatically activated when the date of your birth corresponds to today's date."

* * * * *

It would be wrong to leave the reader with the impression that the particular program elucidated in this paper is typical of what computers can do or are generally used for. The program described here belongs to a subset of activities called <u>simulation</u>, that is, the examination of processes that cannot be handled "analytically" and have to be acted out experimentally (or that are easier handled that way, or that yield analytically more readily if one has experimented first).

It is furthermore in a subset among simulations, namely, the subset of abstract theoretical models rather than more exact, concrete, faithfully descriptive models aimed at "realism" or realistic detail. And among these it represents a very special subset, those that have a two-dimensional interpretation (although we could add dimensions, e.g., by doing floors in a multi-storied building, defining "neighborhoods" and "distances" appropriately).

Finally, an especially simplified version of even this model has been discussed, one in which all groups other than one's own are lumped together as "unlike," and in which local percentages are all that matter. An age-group model would be different: a 40-year-old may consider 30-year-olds "different" but less different than 20-year-olds.

Two features, though, this essay probably shares with any examination of how simulations are programmed. First, "programming" depends on analyzing a rich and complicated phenomenon or process into its elementary steps. Second, there is an inescapable requirement to define the process in exact detail. You can't give a half-articulate instruction to the idiot and wave your arms and say, "You know, ..." He doesn't. An interesting consequence of this discipline is that one often learns much about his own "model" in making the program, even if the computer is never plugged in. (Sometimes he learns that he has to throw it away: there are inconsistencies that can't be programmed, or what appeared to be a new idea embodied in a new model proves, when reduced to an exact statement, to be the same old idea that underlay yesterday's model.)

Annex

The reader may wonder what an actual "program" looks like on paper. There are several different languages for communicating with a computer via teletypewriter; some look a little more like ordinary English than others, but in all of them words like "if" and "go" and "print" and "next" and "for" are actually code names for rather complicated routines the computer has to follow.

The languages all involve a rigid syntax. Some have larger and more flexible vocabularies than others. One language might allow you to say, "Stop if both A and B are true," or "Stop if either A or B is true"; another might permit only the statement, "Stop if A is true." You then have to make the equivalent of the "or" statement by two successive statements, "Stop if A is true," followed by, "Stop if B is true." If B is true but not A, the second instruction will take care of the stopping, with the effect of an "or" statement. The "and" statement could similarly be achieved by two successive instructions: "If A is false, skip the next instruction," followed by, "Stop if B is true."

Below is printed an eleven-line "subroutine." It generates a "random town" of individuals, each identified by his "group number," each occupying a space in a rectangular array.

```
7110  FOR G = 1 TO 3
7120      Z = 0
7130      RANDOMIZE
7140      I = INT(M*RND + 1)
7150      J = INT(N*RND + 1)
7160      IF G(I,J) # 0  THEN  7130
7170      G(I,J) = G
7180      Z = Z + 1
7190      IF Z # Z(G)  THEN  7130
7200  NEXT G
7210  RETURN
```

Lines are numbered. Somewhere in the program there was an instruction, "GOSUB 7110" ("GO to the SUBroutine at Line 7110"), sending the computer to Line 7110. It starts working from there, continuing down the numerical list of instructions until it comes to the word, "RETURN." That is a code word meaning, "Go back, now, to where you came from, when you came in at Line 7110." (If there are several places in the whole program where a "random town" is needed, at each such place there will be a line of instruction saying, "GOSUB 7110"; at Line 7210, getting the "RETURN" signal, the machine must go back to the particular place at which the referral occurred.)

Line 7120 merely introduces a "counting variable." Note that at Line 7180 a +1 is added to Z. The several instructions between 7110 and 7200 are going to be followed repeatedly; and each time the machine runs through the set of instructions it will, at Line 7180, increase the value of Z by 1, thus "counting" the number of times it has run the gamut from 7120 through 7190. In Line 7190 the symbol, #, means "is not equal to," and Z(G) stands for some number; so as long as Z is less than Z(G) it goes back to Line 7130 and plays it all over again. That's what the "IF . . . THEN" instruction does. Z(G) is some number that the machine knows, because we told it earlier what Z(G) stands for. If Z has reached, in its "counting," the value of Z(G), the "IF" condition is not met, so the machine does <u>not</u> go back to Line 7130 but proceeds straight to Line 7200.

Line 7200 ties in with Line 7110, and neither line means anything without the other. Here is what they mean together.

Some variable, known as G, is going to be given successive values, starting with G = 1 and going through G = 2 up to G = 3. That is what the quasi-English phrase, "FOR G = 1 TO 3," sets up. But at each successive value of G, the program that follows (Lines 7120-7190) will be completed. Each time that program has been completed for one of the values of G, Line 7200 says,"NEXT G," and, in conjunction with 7110, that means to step up the value of G to the next number, 2 or 3 as the case may be. Line 7110 halts the process when the program has run with G = 3.

Now, what is the machine doing in between? Well, think of G as the number of "groups" into which the population has been divided. According to 7110 we are going to use three groups. Z(G), down in Line 7190, stands for the number of members there are to be in Group G, a number we put into the program earlier. If there are to be 40 individuals in the first group, Z(1) = 40. Then if there are to be 20 each in the second and third groups, Z(2) = 20 and Z(3) = 20. When G = 1, Z(G) is Z(1) and Z has to count up to 40 before we switch to the next G. Then G changes from 1 to 2; Z is put back at 0 by Line 7120 (or else it would continue counting, "41, 42, ...); then <u>something</u> is done over and over until Z reaches Z(2) = 20; and then "NEXT G" takes us back and we do it all with G = 3.

But what is the "something" being done in there, from 7130 to 7170? What we do is to pick at random one row among the M rows in our rectangular array--one "list" among the M lists that we shall think of as rows in a table--and one column among the N columns in our array--one numbered place on the "list" that gets chosen. We earlier told the machine we wanted M rows and N columns. Suppose we wanted 8 rows and 8 columns; then M = N = 8 and the machine will read Lines 7140 and 7150 as though we had written 8's inside the parentheses where the M and N occur. Line 7140 is full of code words, but what it does is to have the machine pick at random a number from 1 to 8 and call that number I. Line 7150 has the machine pick another number at random, from 1 to 8, and call that one J. Thus, the two lines together pick, at random, two numbers, one of which we call I and let it stand for a row, the other we call J and let it stand for a column.

Now we come to G(I,J) in Line 7170. There are 64 different possible (I,J) combinations if we stick to 8 rows and 8 columns. Each combination has some number associated with it, and the "name" of that number is "G(I,J)." Thus G(3,5) stands for the number associated with the third row and fifth column. For simplicity we begin with G(I,J) = 0 for every value of I and J. We are going to replace the 0's with numbers 1 through 3, and let them stand for the group memberships of the individuals. We can think of the 0 as meaning "unoccupied," if we wish, and our process is to locate individuals, identified by group membership, at empty spaces in the I,J array.

Look at 7160. It says, "If the group number at Row I, Column J is <u>not</u> 0, go back to 7130 and start again." In other words, if there is already an individual at the location denoted by I and J, leave that location and go pick another I and J at random. (Picking at random, we are bound to get repeated "hits" on particular squares; we want to populate only the <u>empty</u> spaces that our procedure turns up for us.) What if G(I,J) is 0? That means that, at the particular I,J location we have picked at random, there is nobody there yet; so we move on to the next step, 7170, and put somebody there.

The way we do this is simple. Remember, G(I,J) is merely a number associated with Row I and Column J. To be concrete, let I = 3 and J = 5 be the two random numbers we just picked. The number associated with Row 3, Column 5 is 0; Line 7170 says to change it so that it is equal to G. What is the value of G? It's 1, 2 or 3, according to whether we are in the process of locating forty 1's, or twenty 2's, or twenty 3's. Line 7170 says to substitute whatever value G currently has for the value that was originally there. If we are still locating the 40 members of Group 1, 0 changes to 1 at Location 3,5. (If we pick a 3 and a 5 at random sometime later in the process, Line 7160 will tell us to go back to 7130 and pick new row and column, because there is a 1 there now.)

So here's the way it goes. We put G equal to 1; we pick two numbers at random, call them Row and Column, and permanently attach a "group number" of 1 to that particular pair of numbers drawn at random. We repeat this until we have attached the number 1 to 40 different formerly empty I,J combinations. At that point Z has been upped 40 times and we go back to 7110, change G to 2, put Z back to 0, and pick I and J values at random until we've hit 20 blank spaces and labeled them with 2's. Then G goes to 3, and we pick more I,J coordinates at random.

By now, with 60 spaces occupied, the chances of picking a blank one are only 4 in 64, so the machine may have to pick a score or more of I,J pairs to find one where G(I,J) = 0. It's speedy, though, and you won't notice the time it takes. It's when we get three 3's located that the situation becomes interesting. There's one blank I,J combination left, and 1 chance in 64 of finding it on a given random try. But after repeated random tries it gets found, and a 3 is located there. There are now forty 1's, twenty 2's, and four 3's located at the 64 different combinations of I,J values. No blank spaces left. What does the machine do now?

IT DOES EXACTLY WHAT WE TOLD IT TO DO. It picks two numbers from 1 to 8 at random, checks its memory to see whether G(I,J) is or is not 0, finds the space occupied and, since G(I,J) # 0, goes back to 7130 and picks two more numbers for I and J. Again it finds the space occupied, and goes back to 7130. And back, and back, and back. Forever.

We gave it an uncompletable task. Not an unperformable one, an uncompletable one. We forgot we had only 64 spaces when we asked for 40, 20 and 20 as group sizes (or we forgot we had 40, 20 and 20 when we told the machine to work with 8 rows and 8 columns). We never programmed the machine to stop when the last blank was filled, because we never intended to fill the last blank. We always planned to have fewer individuals in total than the number of I,J combinations, so we made no plan for the contingency that blank spaces would be exhausted. The machine goes on spinning its wheels forever and ever.

People do make mistakes--mistakes like the one we made above. It may be wise to put in a few safeguards. An easy one is to have the machine add Z(1)+Z(2)+Z(3) and multiply MxN early in the program and, if the sum exceeds the product, to stop and type us a message. Another is to have, along with Z as a counting variable, another one, W, which does not go back to 0, and to tell the machine to stop if W ever gets to some astronomical figure, indicating that something is being repeated insufferably.

The formulae at 7140 and 7150 are full of code symbols; but if you've read this far you may as well hear what they do. RND is a single instruction and tells the machine to find us a random number between 0 and 1, i.e., a random fraction. The asterisk is the multiplication sign, so M*RND tells the machine to multiply M, the number of rows, by some random fraction; the result will be a number in the range from 0 to M, or 0 to 8 if we have 8 rows. By adding 1 we convert that to a number from 1 to 9. The symbol INT( ) means to take the "integer value" of whatever is in the parentheses, the integer value being what's left when everything after the decimal point has been dropped. Thus INT(4.7332) is equal to 4. So by taking just the integers we end up with randomly selected integers from 1 to 8.* (The word, "RANDOMIZE," in Line 7130 tells the machine to pick a new set of random numbers every time we use the program, not to pick a random selection the first time and remember it.)

The name of the language in which Lines 7110-7210 are written is BASIC.

*There is a very small chance that a random number selected by the computer will be exactly 1. In that case the integer value of (8*1+1) will be 9. We don't want a 9. We can avoid this in various ways; otherwise what can (improbably) happen is that one member of our population will be missing. He'll be located in the nonexistent Row 9. He'll never bother us there, but the "counting" will proceed as though he had been put in a legitimate spot.

The actual program described in this paper runs to something over 200 lines with the general appearance of Lines 7110-7210.