Cellular Models: Modelling Epidemics and Social Segregation

Professor Peter Millican, Hertford College, Oxford

Cellular models provide a powerful and relatively straightforward way of modelling many different phenomena, from crystal growth to biological patterning, and from the spread of disease to social interaction. They can also be fun and even quite exciting! We'll be using *The Turtle System*, which has a number of features that make these models relatively easy to program. And we'll be using the programming language *Pascal*, which has been historically a popular teaching language at university level, often used to illustrate algorithms in university textbooks etc. It is visually a very "friendly" language, and most students find it fairly easy to pick up. You are unlikely to be studying it at school, so this has the advantage of extending your experience, avoiding confusion with your other studies, and ensuring that most students in the class will be starting from a similar level of knowledge.

0. Preliminaries

Before you start, please familiarise yourself with programming using the *Turtle System*. If you have a PC (or a Mac with the ability to run *Windows* in a virtual machine), please download *Turtle System* 14 from www.turtle.ox.ac.uk and put it into an appropriate directory on your machine. If not, you can run *Turtle* directly from the website www.turtle.ox.ac.uk/online/ (though this is less powerful than the downloadable system). Exercises are available on the website to help you familiarise yourself with the system (e.g. try the "self-teach exercises" up to exercise 8). Note also that the various examples used below are available in the downloadable and online systems, through the "Examples" menu, and in particular the sub-menu labelled "cellular models".

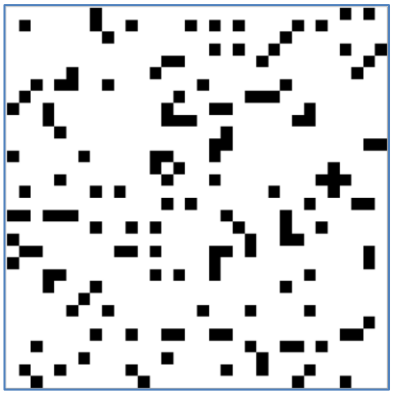
1. Introduction to Cellular Automata, and the Game of Life

A *cellular automaton* is a grid of *cells*, typically in a rectangular array, each of which is in a particular *state* at any given moment. Initially, these states might be assigned randomly or in some pattern, but then as time "ticks" from moment to moment, the state of each cell may change, usually by following simple *rules* that determine the new state according to the arrangement of states across the neighbouring cells. Some cellular automata are *asynchronous*, with individual cells being processed one by one. But the classic type is *synchronous*, with all cells being processed simultaneously, so that with each tick of the clock, we get a new *generation* of cell states, each of which has been individually determined by the same simple rules applied (within its neighbourhood) to the previous generation of states. These rules are standardly *deterministic*, so that the dynamics of the cellular automaton follow inevitably from the starting state. What makes these models so interesting is that even very simple deterministic rules can give rise to surprisingly elaborate patterns of development.

The most famous example of such a cellular automaton is John Conway's *Game of Life*, which involves a rectangular grid of cells – potentially extending for ever in all directions – within which each cell can be either *alive* or *dead* (so there are just two possible cell states). Since an infinite grid is impractical, most computer implementations involve instead a finite square grid, e.g. 50×50, with the edges "wrapping around" (so the right-hand column is treated as being adjacent to the left-hand column, and the bottom row adjacent to the top row). Within this grid, live cells are usually shown black, and dead cells white – the initial arrangement of these black and white cells may be set randomly (e.g. with a 1 in 7 chance of each cell being alive, as in the example below).

When implementing cellular automata within *Turtle*, it often makes sense to change the resolution of the canvas image (as well as the virtual canvas dimensions) so that each cell corresponds to a single "pixel" (as well as a single coordinate location). Then the cell's colour can easily be set using the *pixset* command, as in the following simple Turtle Pascal program which creates an initial setup for the *Game of Life* – an example result is shown in the image.

```
PROGRAM LifeStart;
CONST width=32;
      height=32;
VAR x,y: integer;
BEGIN
 canvas(0,0,width,height);
 resolution(width,height);
 for x := 0 to width-1 do
  for y := 0 to height-1 do
   if random(7)=0 then
    pixset(x,y,black)
   else
    pixset(x,y,white)
END.
```



Use of the two constants *width* and *height* – both here set to 32 – makes this code very easily adaptable for different sizes of board, just by setting the constants to different values. The *canvas* command then specifies the relevant coordinate range (i.e. 0 to 31 along both axes) and the *resolution* command fixes the corresponding image size (i.e. 32×32 pixels). Then the two variables *x* and *y* are used to count through all the cells in turn, randomly making roughly one in seven of them *black* and the rest *white* (the function *random(7)* produces a random integer between 0 and 6 inclusive, so there is a 1 in 7 chance that it will be equal to 0). We see here the pixels' colour being set with the *pixset* command, and the *pixcol* function can be used to read that colour, e.g. `thiscol := pixcol(x,y)`.

The rules that govern the dynamics of the *Game of Life* are as follows. We consider each cell within the grid as having 8 *neighbour* cells, as shown in the picture here. (This differs from the disease model considered in §3 below, in which we shall treat only cells 2, 4, 6, and 8 as neighbours.) What happens to each cell then depends on whether it is alive or dead, and how many of its neighbours are alive or dead. From one generation to the next:

1	2	3
8		4
7	6	5

- A cell that is currently alive will stay alive in the next generation *if, and only if*, it currently has exactly 2 or 3 live neighbours. Otherwise, it dies.
- A cell that is currently dead will become alive in the next generation *if, and only if,* it currently has exactly 3 live neighbours. Otherwise, it stays dead.

Though simple, however, these rules can be tricky to implement, because of the need to perform them on all cells simultaneously. If *A* and *B* are neighbouring cells and we deal with *A* first, then if *A*'s state changes as a result, this risks messing up the calculation for *B*, whose next state should be determined (in part) by *A*'s *current* state, not by *A*'s *next* state. The upshot is that we need to be able to retain in memory *both* the current state of each cell, *and* the newly-calculated state that will take over in the next generation. How this can be done is discussed in another *Turtle* document, but to see such a program in action, run "Conway's Game of Life" from the *Turtle System* "Examples" menu.

2. A Cellular Model of an Epidemic "Tipping Point"

Here we'll be looking at cellular models that are rather easier to implement, partly because they are *asynchronous* and *randomised*, with individual cells being processed in turn rather than entire neighbourhoods. As a first example, load and run the "Tipping point" program (from the "cellular models" sub-menu) within *The Turtle System*. This illustrates the phenomenon discussed in Malcolm Gladwell's 2000 book *The Tipping Point*, pp. 260-1:

> "The best way to understand the Tipping Point is to imagine a hypothetical outbreak of the flu. Suppose, for example, than one summer 1,000 tourists come to Manhattan from Canada carrying an untreatable strain of twenty-four-hour virus. This strain of flu has a 2 percent infection rate, which is to say that one out of every 50 people who come into close contact with someone carrying it catches the bug himself. Let's say that 50 is also exactly the number of people the average Manhattanite – in the course of riding the subways and mingling with colleagues at work – comes into contact with every day. What we have, then is a disease in equilibrium. … With those getting sick and those getting well so perfectly in balance, the flu chugs along at a steady but unspectacular clip through the rest of the summer and the fall. But then comes the Christmas season. The subways and buses get more crowded with tourists and shoppers, and instead of running into an even 50 people a day, the average Manhattanite now has close contact with, say, 55 people a day. All of a sudden, the equilibrium is disrupted. … That moment when the average flu carrier went from running into 50 people a day to running into 55 people was the Tipping Point … at which an ordinary and stable phenomenon … turned into a public health crisis."

The program starts by defining various constants:

(a) Canvas dimensions *width* and *height* (both 100 in this case).
(b) Colours to indicate cells that are *susceptible* (*lightgreen*), *infectious* (*red*), and *recovered* (*blue*).
(c) An initial number of infected individuals (10 by default).
(d) The probability of a contact being infected (to be interpreted as a percentage), and the number of contacts that any infectious individual will have. By default, these are set to 2% and 50 respectively, just as in Gladwell's example.

The point of defining these constants (rather than using numbers directly in the code) is to make them easy to change, so we can experiment with different values, in particular to see how radically the behaviour of the epidemic changes as the number of contacts rises above the "tipping point" of 50.

The program then defines a few variables, notably *numinfected* and *numinfectious*, which will keep track respectively of the total number of individuals who have been *infected* overall, and the number who are currently *infectious*. To make it easy to ensure that these keep in step with the displayed infections, the *infect(x,y)* procedure is defined so as to increment both variables at the same time as a particular pixel changes colour to *infectious*.

When the program runs, random susceptible individuals (i.e. pixels) are selected to be infected until the chosen initial number (c above) has been reached. Then the program main loop randomly selects an individual, and if they are infectious, randomly selects in turn the appropriate number of contacts (e.g. 50 as in d above) and then, with an appropriate probability (e.g. 2% as in d above), infects them if (but only if) they are susceptible. Having gone through all of the contacts, the infected individual now recovers and becomes *recovered* (changing colour to blue). Eventually, the epidemic will subside as more and more individuals become *recovered* (and thus immune). When nobody is left infectious, the program terminates and reports how many individuals have been infected overall.

3. Modelling the Spread of Disease, and Its Prevention

The "Tipping point" program takes no account of the geography of the landscape on which the epidemic is playing out – it just assumes that within the city, everyone is mixing randomly. But now we'll turn to look at a more sophisticated model, in the "Spread of disease" example program. This is one example of an implementation of the well-known "SIR" (Susceptible, Infected, Recovered) model of the spread of infectious disease, and conveys some very important practical lessons about disease prevention. Like the "Tipping point" program, this starts by defining various constants:

(a) Canvas dimensions *width* and *height* (again both 100).
(b) Colours to indicate cells that are *susceptible* (*lightgreen*), *infected* (*red*), and *recovered* (*blue*).
(c) An integer *startradius* (10) that defines the maximum boundary of the initial infection.
(d) Three probabilities, each of which is to be interpreted as a percentage: *infectprob* (1%), the probability that a cell within *startradius* will be initially infected; *immuneprob* (2%), the probability that a cell will be immune throughout (e.g. due to prior vaccination); and *recoverprob* (15%), the probability that an infected cell will recover in any time period.

As before, the point of defining these constants is to make them very easy to change. The whole point of this program, indeed, is to see how the behaviour then changes.

Following the constants and a few variables – including *numinfected*, which keeps count of infected cells – there is again a simple procedure *infect(x,y)*, which colours cell (*x*,*y*) with the *infected* colour (i.e. *red*) and increments *numinfected* accordingly. Then the main program begins, defining the canvas and resolution (just as we saw above), initialising *numinfected* to zero, and colouring the cells of the canvas according to the following rules:

- If a cell's distance from the centre of the canvas is less than or equal to *startradius* (10), then with probability *infectprob* (1%), it will be *infected* (*red*) from the start.
- Otherwise, with probability *immuneprob* (2%), the cell will be *recovered* (*blue*) from the start – this colour is given to cells that are immune, mostly after recovery from the infection.
- Otherwise, the cell will be *susceptible* (*lightgreen*).

The canvas is temporarily frozen (using no*update* ... *update*) while this colouring is taking place, so it can be completed much more quickly.

Having finished this initialisation, the spread and eventual decline of the infection are modelled with a loop that continues until *numinfected* becomes zero. The loop starts by choosing a random value of *x* between 0 and (*width*-1) and a random value of *y* between 0 and (*height*-1). Then we check the colour of cell (*x*,*y*) to see whether it is *infected* (i.e. *red*) or not. If it is, then with probability *recoverprob* (15%), we change it to *recovered* (i.e. *blue*), to indicate that it has now recovered and thus become immune from further infection. (To do this with the correct probability, we use the conditional `if random(100)<recoverprob` to select a random number between 0 and 99 and check whether it is less than *recoverprob*.) Finally, if the cell is infected and has *not* recovered, then the following code is executed:

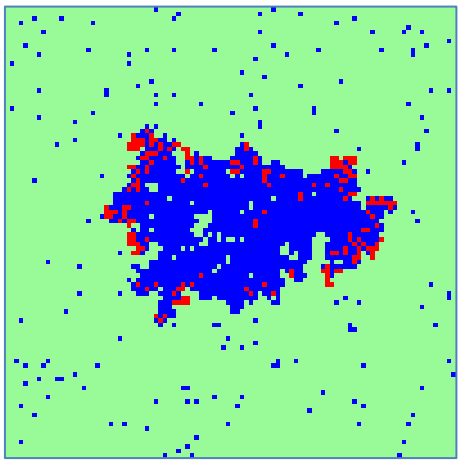
```
n:=random(4)*2+1;
x:=x+n div 3-1;
y:=y+n mod 3-1;
if pixcol(x,y)=susceptible then
 infect(x,y);
```

The purpose of this code is to select at random one of the four closest neighbours of the infected cell and then, if that cell is *susceptible* (i.e. *lightgreen*), to infect it – this is how the infection spreads. The arithmetic here is neat but a bit tricky, going through the following steps and using the operators for integer division (*div*) and remainder (*mod*):

`random(4)`	`random(4)*2`	`n`	`n div 3`	`n div 3 - 1`	`n mod 3`	`n mod 3 - 1`
0	0	1	0	-1	1	0
1	2	3	1	0	0	-1
2	4	5	1	0	2	1
3	6	7	2	1	1	0

The shaded columns show the four possible random numbers (between 0 and 3), and the corresponding values that get added to *x* and *y* respectively. Adding (-1,0) corresponds to a move left on the canvas, (0,-1) to a move up, (0,1) to a move down, and (1,0) to a move right. So after these additions, the coordinates (*x,y*) do indeed identify one of the four neighbouring cells. Now it just remains to test whether that cell is *susceptible* (*lightgreen*), and if it is, to infect it (*red*).

> You might have noticed that moving in these ways from a cell on the edge of the grid could take us to a pixel off the canvas. One convenient feature of *pixset* and *pixcol* – unlike corresponding operations with arrays – is that they do not throw up an error message if this happens, and since it causes no trouble for the operation of our program, we are able to ignore this complication here. But an inquisitive student might want to investigate whether changing the colours *could* introduce a problem …



One practical virtue of this model of infection – which in its more sophisticated forms is highly influential and widely used – is to demonstrate very clearly the value of vaccination. If the program is run as it stands, the infection is very likely to spread from the centre of the canvas to most of the susceptible cells (the image here shows it spreading aggressively in several directions). But if the value of *immuneprob* is set higher – for example, changed from 2% to 12% – then you will find that the infection has far less impact, often dying out quickly and usually reaching only a small proportion of the canvas. Thus artificially vaccinating even 10% of the population can potentially bring a huge payoff in disease control for the population as a whole. Real diseases, of course, will vary in infectivity and other characteristics, so we cannot assume that this conclusion will apply to them. But this model does allow for variation, and enables us to explore how the critical value of *immuneprob* at which the disease can be tamed depends on the probability of recovery in each time period: if recovery typically takes a long time (because *recoverprob* is low), then more widespread prior immunity will be required to keep the disease in check. The crucial point here is that the longer recovery takes for any individual, the more opportunity the disease has to infect that individual's neighbours, and so the higher the probability that it will indeed be passed on to them.

This particular population structure – which assumes that every individual is statically located within a fixed grid, with precisely four neighbours each – is of course very crude, but more complex versions of the "SIR" model play a vitally important role in the real world, helping *epidemiologists* (those who study such things) to understand, predict and combat the spread of diseases. In the mathematics of infection, the most important parameter for any disease is its "basic reproduction number" (commonly denoted R_0) – that is, the number of individuals that could be expected, on average, to be directly infected from one newly-infected individual within a totally susceptible population. In our model (with *recoverprob* at 15%), this parameter is around 2.35 for an individual surrounded by four "susceptibles", but of course any individual thus infected will then have only three adjacent susceptibles (since the individual who infected them is no longer susceptible), and their expected direct infectivity drops accordingly, to around 1.76.

An interesting experiment is to see what happens if individuals are allowed to wander (e.g. perhaps by incorporating some sort of "diffusion", or long-range swaps – as a crude model for air travel), in which case the spread of disease is likely to be significantly greater. The "Spread of disease" program actually allows for this, by providing a constant, *movement*, which is set to *false* by default, but can be edited to *true* if you want to see the effect. Another constant, *report*, can be made *true* if you want to see updates on the progress of the epidemic.

4. Schelling's Segregation Model

Now we go on to examine in detail another built-in program from the *Turtle System* (again under the "cellular models" Examples sub-menu), but this time one of social rather than medical relevance. This implements a model of social segregation due to the Nobel prizewinning economist Thomas Schelling, which is famous for showing how relative small local preferences can lead to global segregation. If you run the program and then quickly click on "HALT" (before 2 seconds have elapsed), you will see the initial situation in which we have a "map" of a square city surrounded by a rim of *green*. Within the city, most of the cells are either *blue* or *red* – in a strictly alternating pattern – but roughly 4% (1 in 25) are *green*. The *blue* and *red* cells represent dwellings that are occupied by members of two different religious communities, the Blues and the Reds, while the *green* cells are unoccupied. Now run the program again, and see what happens, as the Blues and Reds who find that they are "unhappy" with their neighbourhood move around. The precise outcome depends on chance, but almost invariably, you will end up with a pattern in which the Blues and the Reds are very largely segregated rather than interspersed. You might naturally think this indicates that they are strongly biased against the other group, but in fact this result comes about even when the bias is relatively mild.

As with the Game of Life, we consider each cell as having 8 *neighbour* cells. To assess any individual's "happiness", we simply count the number of neighbours that have the same religion (variable *like*) and the number of neighbours that have the opposite religion (variable *unlike*), and check whether it is true that ($like \geq unlike-1$). If it is true, then the individual is "happy"; if false, "unhappy". In other words, an individual becomes unhappy when the neighbours from the *unlike* religion outnumber the neighbours from the *like* religion by 2 or more. So, for example, a Red individual with 3 Red neighbours will be happy as long as they have a maximum of 4 Blue neighbours; but with 3 Red and 5 Blue neighbours, they would become unhappy. Any individual assessed as unhappy will immediately try to find an *empty* (i.e. *green*) cell where they would be happy, and on finding one, will move there (and the cell from which they move will become *empty*). This one simple rule governs the entire dynamics of the model! Let us now go through the program, line by line:

```
PROGRAM schelling;
CONST width=50;
      height=50;
      empty=green;
VAR test,tryi,tryj: integer;
    this,i,j: integer;
```

As is usual with these models, the *width* and *height* of the grid are set as constants, to make the code easily adjustable, and the constant *empty* is set as *green*, which makes that easily changeable also (e.g. to *midgrey*) if desired. Then a number of integer variables are declared, whose use we shall see below in the main program (after *BEGIN*). First, however, we come to a *subroutine* of a special kind ...

```
Function happy(x,y,c: integer): boolean;
```

This indicates the definition of a *function*, which will prove very useful in the rest of the program. It is declared here as a *boolean* function, meaning that it can return one of the values *true* or *false*. As the name of the function ("happy") indicates, it is intended to return the value *true* if an individual of colour *c* would be happy at location (*x*, *y*), and *false* if that individual would be unhappy there.

```
Var like,unlike,neighbour,i,j: integer;
```

This line declares five integer variables for use within the function. Note that a function – just like a procedure – is a sort of "mini-program" that can have its own "local" variables. It also has its own *Begin* and *End*, which enclose the function's commands:

```
Begin
 like:=0;
 unlike:=0;
```

These lines initialise the count of *like* and *unlike* neighbours for the "current" cell at location (*x*, *y*). Both counts, obviously, start at 0.

```
 for i:=-1 to 1 do
  for j:=-1 to 1 do
   if (i<>0) or (j<>0) then
```

Variables *i* and *j* are used, respectively, for the *x*-offset and *y*-offset of each neighbour from the current cell (*x*, *y*), so the neighbour in question will be cell (*x*+*i*, *y*+*j*). These three lines ensure that we count through all eight neighbours of the current cell, making each of *i* and *j* equal to -1, 0 and 1 in turn, but ignoring the case in which both *i* and *j* are 0 (since this would be the current cell itself).

```
    begin
     neighbour:=pixcol(x+i,y+j);
     if neighbour<>empty then
      begin
       if neighbour=c then
        inc(like)
       else
        inc(unlike)
      end
    end;
```

These are the commands performed in turn for each of the eight neighbouring cells. First, *neighbour* is made equal to the relevant pixel colour. If this is *empty* (i.e. the relevant neighbouring cell is a green space), nothing further happens. Otherwise, *neighbour* must be either *blue* or *red*, meaning that

either *like* or *unlike* should be incremented, depending on whether it matches colour *c* (which is the colour of the individual whose happiness – or potential happiness – we are trying to assess). The upshot of all this is that after having gone through this code eight times (once for each neighbouring cell), *like* will record the number of neighbouring cells of colour *c*, and *unlike* will record the number of neighbouring cells of the "opposite" colour.

```
 result:=(like>=unlike-1)
End;
```

These two lines specify the *result* of the function (i.e. *true* if (*like* ≥ *unlike*–1) and *false* otherwise), and then end the function definition. Our function *happy* has now been defined, and can be used in the remainder of the program, to which we now turn.

```
BEGIN
 canvas(-1,-1,width+2,height+2);
 resolution(width+2,height+2);
```

BEGIN indicates the start of the main program, after which the Canvas size (i.e. coordinate range) and resolution (i.e. number of pixels) are specified. Here the coordinates will range from -1 to *width* in the *x* direction (with *width*+2 horizontal pixels) and -1 to *height* in the *y* direction (with *height*+2 vertical pixels). So for example if *width* has been set to 50, there will be 52 horizontal pixels, and each of these will correspond to a distinct *x*-coordinate, from -1 to 50 inclusive. (Likewise for *height* and the *y*-coordinates.) The reason for the "+2" is to allow for an *empty* (*green*) border, one cell wide, around the 50x50 city map, which saves us from worrying about the edge cells within the city.

```
noupdate;
blank(empty);
for i:=0 to width-1 do
  for j:=0 to height-1 do
```

The *noupdate* command prevents the Canvas display from updating until the initial disposition has been completed, so as to make the start of the program neater and quicker. Then *blank*(*empty*) blanks the entire Canvas to the colour *empty* (i.e. *green*), thus creating the green borders. The double loop that follows will count through every cell (*i*, *j*) within the city (*i* counting horizontally from *0* to *width*-1, and *j* counting vertically from 0 to *height*-1), setting the initial colour of each cell.

```
   begin
    test:=random(25);
    if test=0 then
     pixset(i,j,empty)
    else
     begin
      if (i+j) mod 2=0 then
       pixset(i,j,red)
      else
       pixset(i,j,blue)
     end
   end;
 pause(2000);
```

As the first command within the loop, *test* is set to a random number between 0 and 24; if this number is 0 (i.e. with a 1-in-25 chance, or 4% probability), the pixel at location (*i*, *j*) is set to *empty* (i.e. *green* – which in fact it is already). Otherwise, the relevant pixel is set to *red* if (*i*+*j*) is an even number (i.e.

its remainder on division by 2 is zero) or *blue* if it is an odd number. Hence the entire city is coloured in a "chequerboard" *red*/*blue* fashion, except for the 4% *green* cells. That done, the program *pauses* for two seconds (2000 milliseconds) to show this idyllic scene of (temporary) social integration!

```
while ?key<>\escape do
 begin
  noupdate;
```

The *while* loop here makes it possible to halt the program by pressing the *Escape* key. The loop that follows will combine four tasks:

- Find a cell whose occupant is currently unhappy, and record their colour as *this* (note that we are now using the variables declared right at the start of the program).
- Colour the relevant cell *empty* (i.e. *green*), to show that the occupant has left.
- Find a cell which is currently *empty*, and where an occupant of *this* colour would be happy.
- Colour the relevant cell *this* colour, to show that it is now occupied.

The loop starts with *noupdate* – and will finish with *update* – to ensure that the switching of colours is done smoothly, with both recolourings becoming visible at the same time.

```
  repeat
   tryi:=random(width);
   tryj:=random(height);
   this:=pixcol(tryi,tryj)
  until (this<>empty) and (not(happy(tryi,tryj,this)));
  pixset(tryi,tryj,empty);
```

As noted, we are using variables declared at the start of the program. First *tryi* is set to a random value between 0 and *width*-1, and *tryj* is set to a random value between 0 and *height*-1: these together mean that the coordinate pair (*tryi*, *tryj*) specifies a random cell within the city grid. Then *this* is set to the pixel colour of that cell, and this whole sequence is repeated until we have found a cell which is *not empty*, and whose occupant is *unhappy* at that position. Having found such a cell, it is made *empty*, and we move on to finding a new location for the previously unhappy occupant.

```
  repeat
   tryi:=random(width);
   tryj:=random(height)
  until (pixcol(tryi,tryj)=empty) and happy(tryi,tryj,this);
  pixset(tryi,tryj,this);
```

Again we repeatedly find a random cell with coordinates (*tryi*, *tryj*) within the city grid, but this time the sequence is repeated until we have found a cell which is *empty*, and where the individual we are trying to move (whose colour is *this*) would be *happy*. Having found such a cell, it is coloured *this* to complete the individual's move from an unhappy to a happy location.

```
   update
  end
```

As remarked earlier, the loop finishes with *update* to ensure that the recolourings (of the moved individual's old and new cells) take place at the same time.

```
END.
```

The program ends after the *while* loop finishes (i.e. when the *Escape* key is pressed), or the "HALT" button is clicked.

5. Some Ideas for Further Exploration

Many different variations are possible on the models we have considered here, and you can have a lot of fun – and also learn a lot about programming – by trying to find interesting ways to extend or modify them. To get you started, here are some suggestions.

(a) Epidemic Programs

1. In the "Tipping point" program, there is no limit on the geographical distance between infectious individuals and their contacts – they are all just assumed to be mixing randomly within a city's transport system. You might like to see what happens if you impose such a limit, or if – for example – you insist on the vast majority of an individual's contacts being within the same quadrant of the city.

2. The box on page 5 above suggests that "an inquisitive student might want to investigate whether changing the colours *could* introduce a problem" into the "Spread of disease" program. Can you discover what sort of problem this is hinting at?

3. Within either of these programs, you might want to try modifications inspired by the Covid epidemic – e.g. implementing partial "lockdowns", or travel bans, or tests for those who suspect that they may be infected followed by "self-isolation". You could also try adapting the programs so that you are able interactively to "infect" one of the cells by clicking on it, thus simulating the effect of someone infected arriving from abroad.

4. More ambitiously, you might like to consider developing a program that allows for virus mutation, with people who have recovered being only partially immune to new variants, depending on how much the virus has changed in the meantime. (For example, you might colour individuals with different shades of red, depending on the virus concerned.)

(b) Schelling Segregation Model

5. Currently the program defines a "neighbourhood" as 8 cells: a 3x3 square omitting the central cell. Can you adapt it to use a neighbourhood of 24 cells (i.e. 5x5 omitting the central cell)? If you do this, what "happy" rule would give the best results?

6. The "happy" rule currently depends on the absolute difference between *like* and *unlike*, irrespective of the total number of neighbours. Can you make it depend instead on the relevant *ratios* (e.g. an individual might be happy if *unlike* is less than 150% of *like*, or equivalently, less than 60% of the total number of neighbours)?

7. Sometimes the program seems to stop even when there are still some unhappy individuals who could be made happy by moving (something you will see far more often if you change "random(25)" to "random(100)" at line 37). Can you see why this might be, and can you suggest a way of fixing it, either totally or partially?

8. Imagine that the city is being settled, from the start, by alternate Blue and Red individuals moving into random cells where they would be happy. How does this affect the dynamics?