The Travelling Salesman

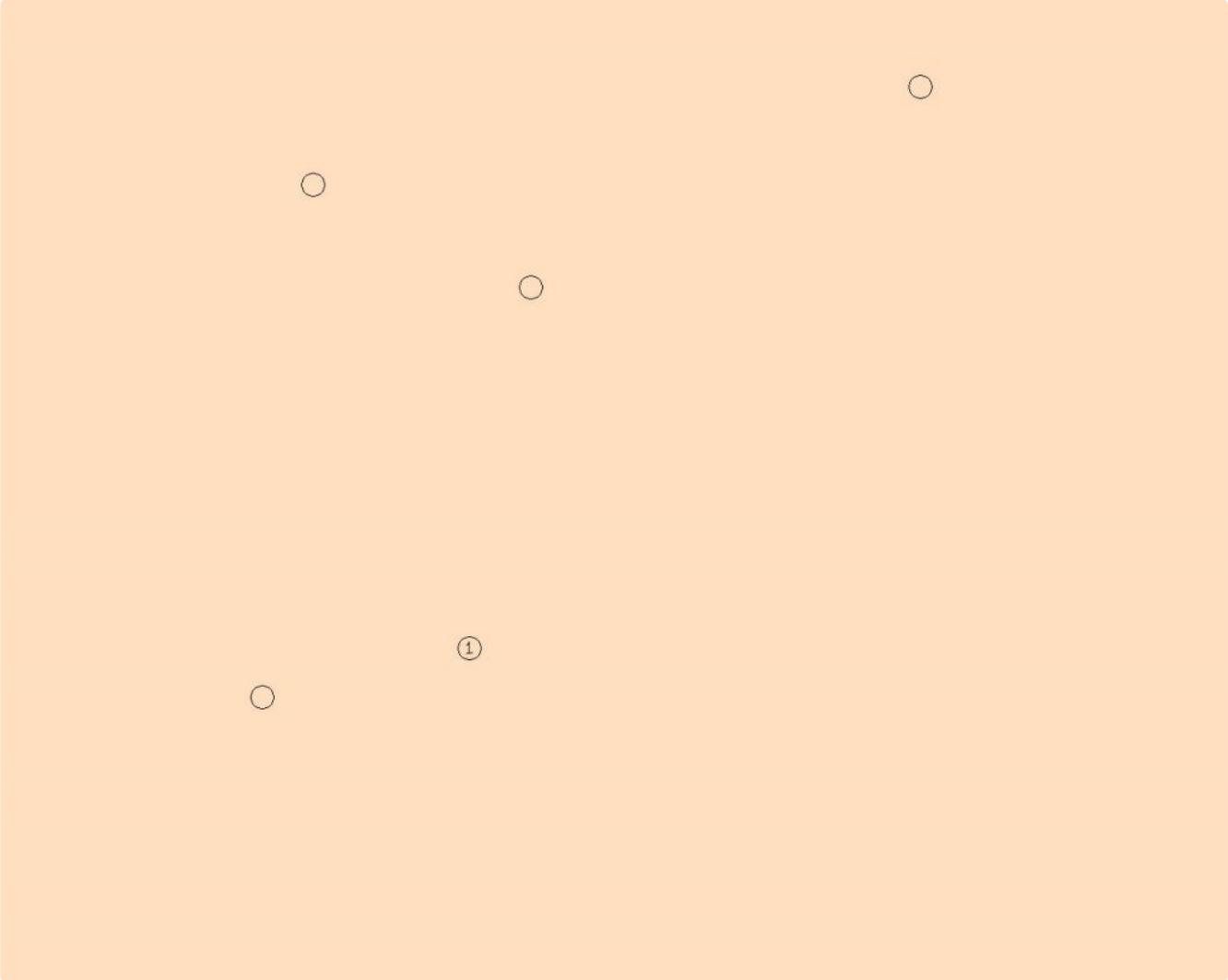
Our goal in this exercise is to find a reasonably short route through an arbitrary number of cities splattered, randomly, on a map (or in this case, on the screen). We begin with just two Plain English type definitions:

```
A city is a thing with a number, a spot and a visited flag.
A nearest neighbor is a city.
```

Then we create a list of five cities at random locations on the screen using this routine:

```
To create some cities:
Put 5 into a number.
Make a box 1 inch smaller than the screen's box.
Loop.
Pick a spot anywhere in the box.
Allocate memory for a city.
Put the spot into the city's spot.
Append the city to the cities.
Subtract 1 from the number. If the number is 0, break.
Repeat.
Put 1 into the cities' first's number.
```

The starting city is at the top of the list and is marked with the number "1". When we draw those cities on the screen they look like this:



Then we arrange those cities into a reasonable sequence for the travelling salesman. We do this using the Nearest Neighbor algorithm. It doesn't guarantee the shortest possible path, but it is very simple and very fast and produces a *reasonably* short path through the cities. Great engineering seeks a balance between competing objectives.

This is the Plain English routine that implements the Nearest Neighbor algorithm:

```
To arrange some cities for a travelling salesman (nearest neighbor algorithm):
Get a city from the cities.
If the city is nil, break.
Set the city's visited flag.
Find a nearest neighbor to the city in the cities.
If the nearest neighbor is nil, break.
Remove the nearest neighbor from the cities.
Insert the nearest neighbor into the cities after the city.
Repeat.
Number the cities.
Draw the cities.
```

Finding the nearest neighbor of a city requires calculating the distance between the current city and all of the other cities, which is accomplished using this routine:

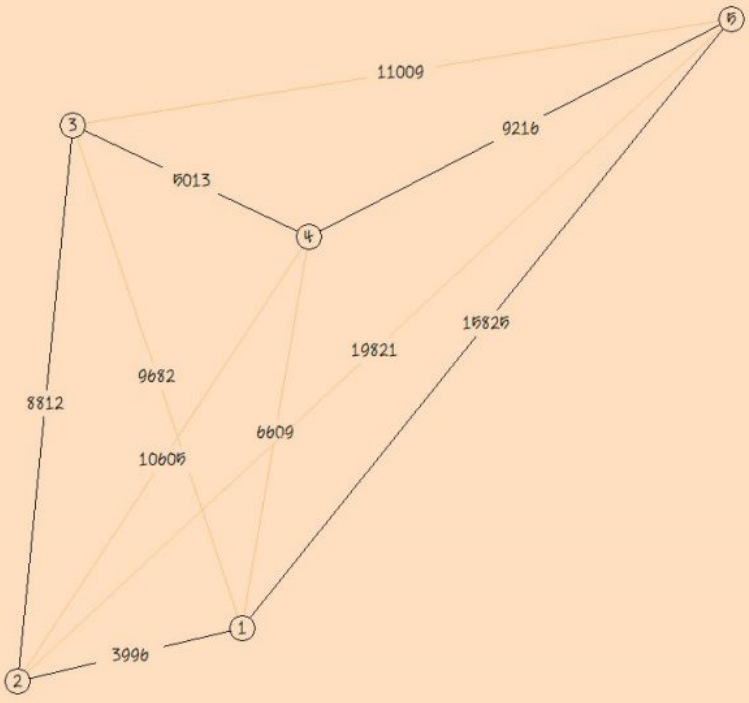
```
To find a nearest neighbor to a current city in some cities:
Void the nearest neighbor.
Put the largest number into a shortest distance.
```

```
Loop.
Get a city from the cities.
If the city is nil, break.
If the city is the current city, repeat.
If the city's visited flag is set, repeat.
Get a distance between the city and the current city.
If the distance is not less than the shortest distance, repeat.
Put the distance into the shortest distance.
Put the city into the nearest neighbor.
Repeat.
```

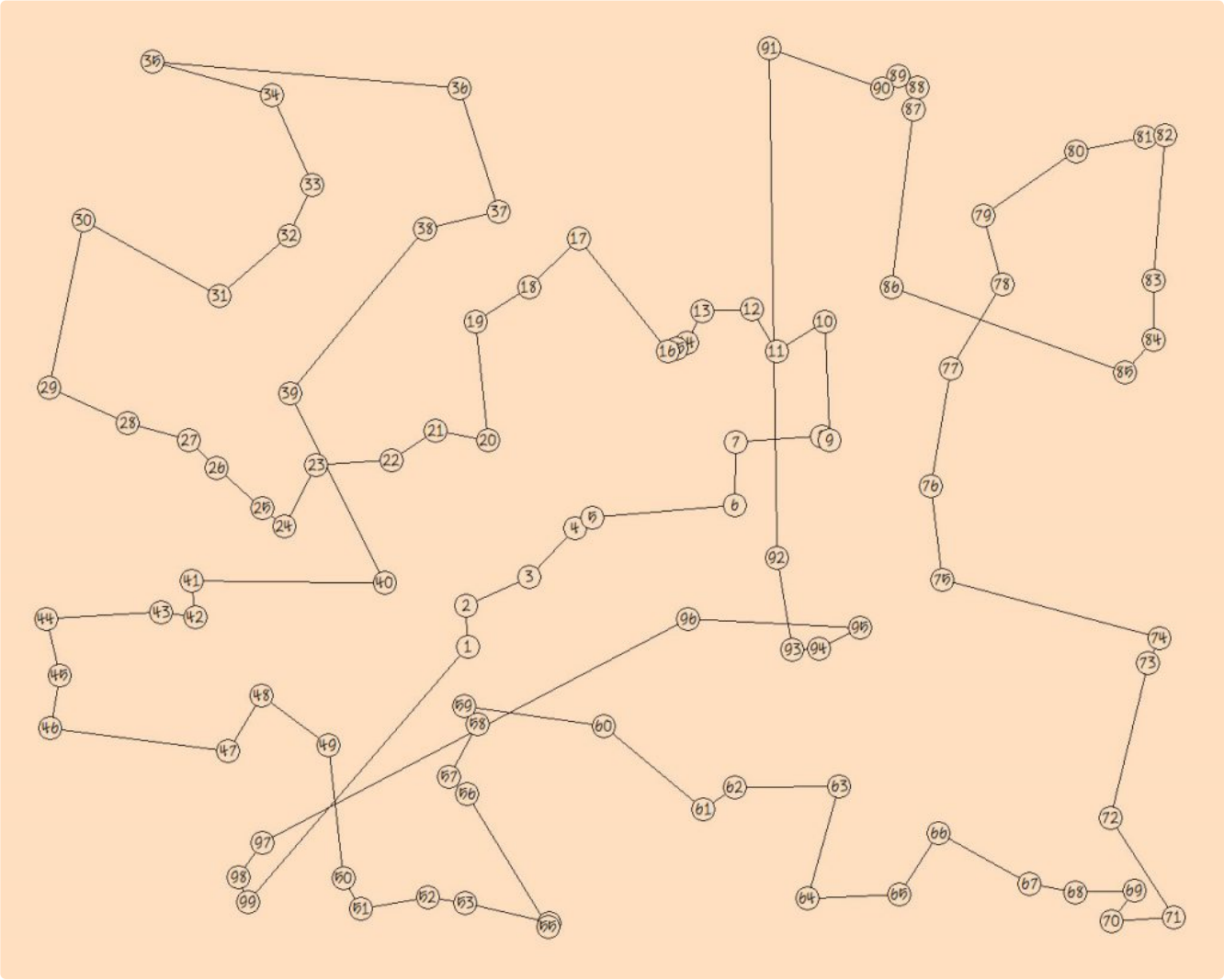
Since we Osmosians don't believe in transcendental functions (like *sine* and *cosine*), we calculate the Minkowski distance between the cities using this simple and speedy routine:

```
To get a distance between a spot and another spot (minkowski):
Put the spot's x minus the other spot's x into a number.
De-sign the number.
Put the spot's y minus the other spot's y into another number.
De-sign the other number.
Put the number plus the other number into the distance.
```

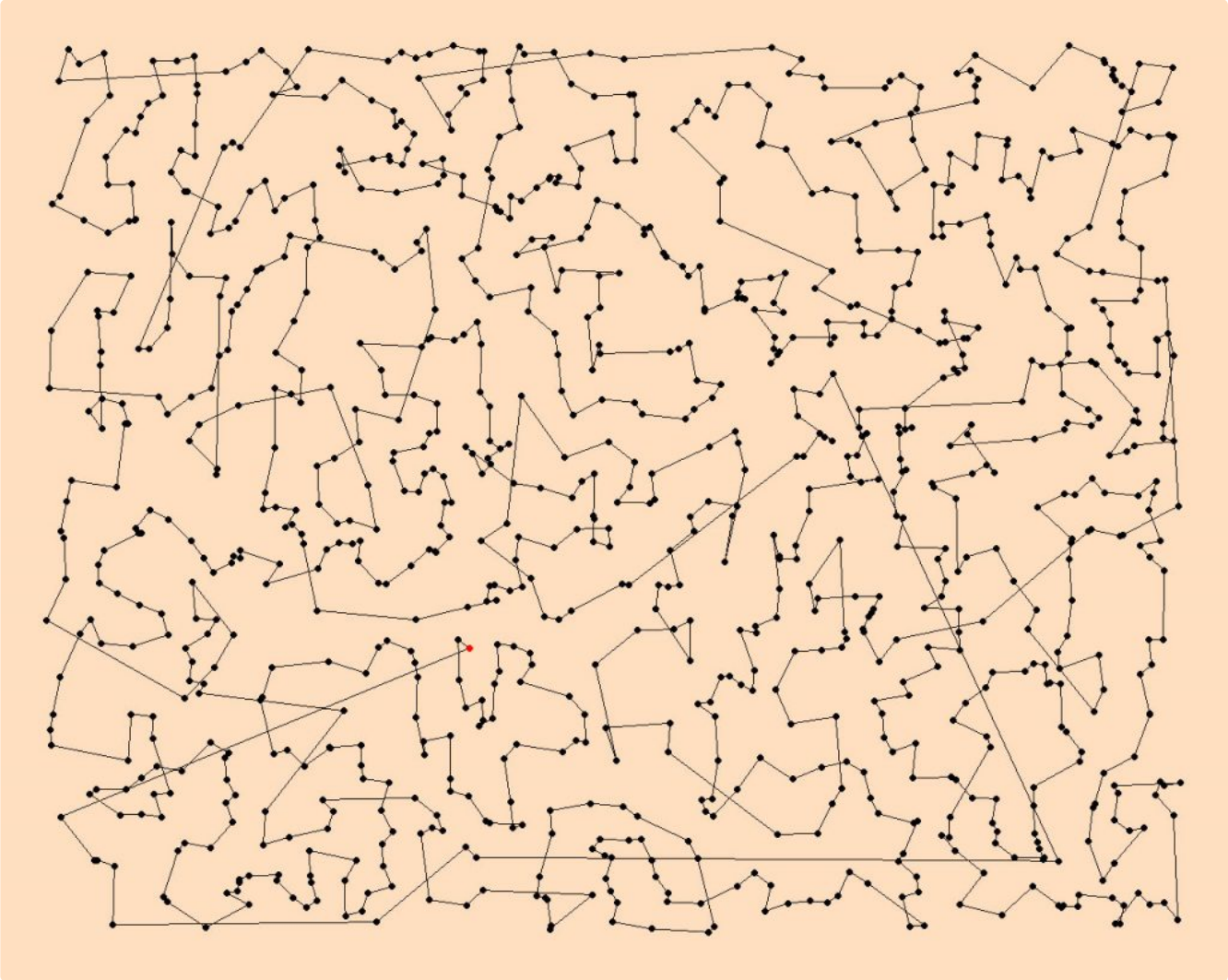
Once the cities are arranged in a proper sequence, we can redraw the map like this:



And that's about all there is to it. But before we go, let's see what it does with 99 cities (the biggest number that will fit inside a city circle on the map). I eliminated the intermediate lines and the distance numbers for this test to keep the screen from getting too cluttered:



Less than a second to run, and a pretty good path. I also tried it with 1,000 cities (and smaller city markers, the starting city marked in red). It still took less than a second…



...but you can see it got lost a couple of times. But who knows? Maybe our salesman likes a long, quiet drive now and then, to collect his thoughts or listen to the radio.