The following instructions apply to a Unix-based machine. If you have a Windows machine you may want to download Ubuntu or a similar implementation, or use the VCL facility (vcl.ncsu.edu).
After unzipping, simply do `make`.
To generate a graph, do
`./nextgen` *`type vertices edges seed > output-file-name`*
*type* is `random` (edges selected completely at random) or `geometric` (vertices are integer points inside a large square, connected by an edge if they are within a given distance of each other; distance is calculated so as to approximate the desired number of edges), or geo_wrap (vertices near the boundary of the square are connected with ones near the opposite boundary – a wraparound)
*vertices* and *edges* are the number of vertices and edges, respectively
*seed* is a random number seed (any integer will do and it need not be the same in both places).

Weights are random for random graphs and based on distance for geometric graphs. Distance in geometric graphs is computed using the "infinity norm": distance between $(x_1, y_1)$ and $(x_2, y_2)$ is $\max(|x_1-x_2|, |y_1-y_2|)$. The weights can be very large but are guaranteed to be small enough so that the total weight of a minimum spanning tree or a path will never exceed `LONG_MAX`, the size of the maximum positive integer on most machines $(2^{31} - 1)$.

Instead of sending the output directly to a file, you can filter it first to put edge weights in a given range. If you do
`./nextgen` *`type vertices edges seed`* `| ./normalize_weights` *`max_weight > output-file-name`*
you will get a graph whose edge weights are integers in the range $[1, max\_weight]$.

Geometric (and geo_wrap) graph generation is problematic in two ways: (i) the number of edges in the generated graph can be much different from what was requested; and (ii) the generated graph may not be connected. Output to stderr gives information about actual number of edges and number of components both before and after the program adds extra edges in an effort to mitigate (ii).

The script `generate.sh` (run it with no arguments for instructions) calls `nextgen` and performs three useful tasks: (a) the number of edges in the file header for geometric and geo_wrap graphs is corrected to reflect the true number of edges; (b) generation is terminated if a geometric or geo_wrap graph is not connected; and (c) the output is sent to a file with a standardized name that reflects how the graph was generated. The graph types for the script are abbreviations: `rn` for random, `gm` for geometric, and `gw` for geo_wrap.

Graphs have the following format (called `gph`):
*Any number of lines having* `c` *in the first column*
`g` *vertices edges*
*if graphs are based on geometry, any number of lines of the form*
`n` *x-coordinate y-coordinate*
*any number of lines of the form*
`e` *vertex-one vertex-two weight*
The *edges* in the `g` line is only an estimate in the case of geometric graphs, but will be exact if the `generate.sh` wrapper is used. Each `e` line represents an edge between two numbered vertices.

***Caution:*** *Vertices are numbered starting at* 1. *So, if you're using arrays in C or Java (or many other languages), ignore the entry at index* 0 *and allocate for* $n + 1$ *elements instead of* $n$

Note: The graphs generated by nextgen are guaranteed to be connected but the means for accomplishing this are not necessarily "sound" in the sense that all connected graphs of the given size are equally likely.

Additional useful programs are listed below. For each of them you can get a brief usage message by running the program with no command-line arguments. With the exception of `random_delaunay.py`, the programs are text filters, taking a graph file in the above format from standard input and printing a modified graph file to standard output.

- `permute-edges`: output is the input graph with the list of edges randomly permuted.
- `normalize-weights`: output is the input graph with weights changed so that they are integers ranging from 1 to a specified number.

- `../scripts/gph2graphml.py`: converts a `gph` file to the `graphml` format used by `Galant`; if node positions are not specified in the `gph` file they are assigned randomly; there are optional arguments – use the `-h` option to find out more.
- `random_delaunay.py`: produces a Delaunay triangulation with $n$ vertices, where n is given; the triangulation is based on $n$ randomly chosen integer points in an $M \times M$ square, where $M$ is half of the maximum possible integer (half because otherwise distances between points might be larger than the largest integer). There are several options (use `-h` to get a list), including the ability to create duals of the resulting planar graphs. In order to get graphs in the desired format, use `-f gph`.