Generating Strongly Connected Random Graphs

Peter M. Maurer
Dept. of Computer Science
Baylor University
Waco, Texas 76798

Abstract – The algorithm presented here is capable of generating strongly connected graphs in a single pass that requires O(n) time. The method is to create a spanning tree that is a directed acyclic graph, and adding a minimal number of edges to make the spanning tree strongly connected. This is done in a way that is completely general. Once the strongly connected spanning tree is created, additional edges can be added to the tree to create an arbitrary strongly connected graph.

1 Introduction

One important problem in many types of simulation is creating random data for input to the simulation. Over the years, our need for such data in the simulation of gate-level circuits has led us to create a package for creating many different types of random data [1,2]. Despite its utility, this package has become dated. It was originally intended to generate random data files for input to a simulation program. Recently, we have begun a project to upgrade this package with new features to make it more useful for modern types of programs that do not depend heavily on file-based input, and to generate types of data that are more suitable to modern programs than character strings and simple binary values.

One major focus of this activity (there are many) is the generation of random graphs. Graphs can be used to model many real-world phenomena. There are too many applications to mention individually, but see [3] for an example. One new feature of our package is the creation of graph-generation subroutines that can be incorporated into existing software. The graphs are generated internally as adjacency lists and passed, as pointers, to the simulation software.

Graph specifications are simple, typically one line, but permit the specification of many different types of random graphs. The most common models are the edged-oriented models, the Gilbert model [4] and the Erdős–Rényi model [5]. The vertex-oriented models, power-law [6] and degree-sequence [7] models are also fairly common. This paper will focus on the edge-oriented models. The Gilbert model assigns a probability of p to the existence of any edge, and the Erdős–Rényi model assigns equal probability to all graphs with M edges. For the Gilbert model we generate k vertices and add each edge from the complete graph with probability p. For the Erdős–Rényi model, we sort all edges into random order and choose the first M edges from the sorted list. (The parameters k, i, and M are specified by the user.) The power-law and degree-sequence models are also available in our package, but these are beyond the scope of this paper.

Parameters can be used to specify that the graph is directed, or that the graph must be connected, or both. Creating a connected non-directed graph is relatively simple. We generate a spanning tree using Algorithm 1, and then apply either the Gilbert or the Erdős–Rényi model to the remaining edges. The purpose of Algorithm 1 is to create a spanning tree for the graph. A non-directed graph is connected if and only if a spanning tree exists. Algorithm 1 adds Vertex 0 to the spanning tree, and then adds the remaining vertices by selecting a random vertex from the partial spanning tree as the parent of the new vertex.

1. Add Vertex 0 to the tree.
2. For each vertex, *i*, 1 through $k-1$
   a. select a vertex *j* at random from the existing tree vertices
   b. Add an edge between *i* and *j*.
   c. Add vertex *i* to the tree.

Algorithm 1. Creating the Spanning Tree.

When the graph is directed, simply creating a spanning tree is insufficient because the resultant graph must be *strongly* connected. The spanning tree is still necessary, because there must be a path from Vertex 0 to every other vertex. When creating a spanning tree for a directed graph, the first step is to modify step 2 of Algorithm 1 so that the new edge proceeds from the tree vertex to the new vertex. This insures that there is a path from Vertex 0 to every other vertex. The resultant tree is a directed acyclic graph with the root of the tree as the only source. The sinks are the leaves of the tree.

There are several straightforward methods for making the spanning tree strongly connected. When adding a tree vertex, we could add two edges, one from the tree vertex to the new vertex, and another in the opposite direction. This would make the tree strongly connected, but there would be no way to generate certain types of graphs such as the simple cycle of Figure 1. Another method is to add an edge from each leaf vertex to the root vertex. This is perhaps more general, but graphs such as that pictured in Figure 2 would be impossible to generate.

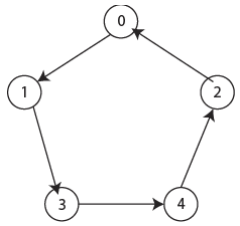

Figure 1. A Simple Cycle.

ISBN: 1-60132-465-0, CSREA Press ©

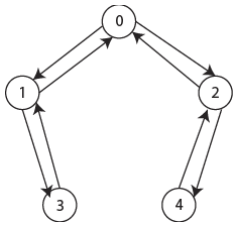

Figure 2. Impossible for Leaf-to-Root.

It is necessary to be able to generate *any* strongly connected graph on k vertices. Some attempts have been made to do so using the rejection method [8]. (The rejection method repeatedly generates directed graphs at random, and rejects those that are not strongly connected.) However, the rejection method works well only if the probability of rejection is small. When a sparse strongly connected graph is required, the rejection method is not suitable.

Creation of the spanning tree is clearly the starting point for such a procedure, since it must exist. Furthermore, Algorithm 1 is able to generate any spanning tree, up to isomorphism. The next step must be to add additional edges to the graph to make it strongly connected. Furthermore, this should be done in a way that is certain to succeed and adds only a minimal number of edges to the tree. (The Leaf-to-Root method generates the absolute minimum number of edges, but is not suitable because it is not able to generate all strongly connected graphs.)

To meet the needs of our generation software, we use a modified version of the Tarjan strongly connected component algorithm [9]. The Tarjan algorithm is capable of identifying any strongly connected graph by performing a single depth first search. The forward edges of the depth first search define a tree which is equivalent to our initial spanning tree. To detect a strongly connected graph, the Tarjan algorithm identifies a minimal set of back and cross edges to insure that the initial spanning tree strongly connected. Rather than detecting such edges (which do not exist in the initial spanning tree) we modify the algorithm to insert such edges where required. We do this in such a way that any suitable set of back or cross edge can be generated. Once the tree has been made strongly connected the Gilbert or Erdős–Rényi model can be applied to the remaining edges.

2 The Tarjan Algorithm

To understand our method of inserting edges into the tree it is necessary to understand the principles of the Tarjan algorithm. The mechanism is based on depth first search with computed start times. Algorithm 2 shows the basic depth first search algorithm with the modification points that are used to detect strongly connected components. Start time is an integer in the range $[0, n-1]$ where n is the number of vertices in the graph. The start time of vertex i is a sequential number indicating the order in which vertex i was first seen by the depth first search algorithm. In Algorithm 2, each element of the *StartTime* array (which is of size n) is initialized to -1, and *GlobalStartTime* is initialized to zero. Both are global variables.

1. Function call DFS(i)
2. Set StartTime[i] to GlobalStartTime. // vertex i is first seen
3. Increment GlobalStartTime by 1.
4. For each vertex j adjacent to vertex i
    a. If StartTime[j] is equal to -1 Call DFS(j) // back up to i
5. // backing up from i

Algorithm 2, basic DFS.

To detect strongly components, the basic DFS algorithm must be modified in three places. First, we add an array of size n named *Low*. After a vertex has been completely processed, it *Low*[*i*] will contain the smallest start time of any vertex that can be reached by following zero or more tree edges from Vertex *i*, followed by one back edge or cross edge. The following is added after step 2.

2.5. Set Low[i] equal to StartTime[i].

This step indicates that, initially, the lowest reachable start time is the start time of the current vertex. To step 4a we add *Low*[*i*] = min(*Low*[*i*],*Low*[*j*]). If it is possible to get further back than the current value of *Low*[*i*] by using tree edges, the new minimal start time recorded.

The following step is added after step 4a:

b. else if j is not already in a SCC, Low[i] = min(Low[i],StartTime[j])

If the edge (i,j) is a cross edge or a back edge, and it is further back than we have been able to get previously, its start time is recorded. The only problem that arises is when (i,j) is a cross edge to a strongly connected component that has already been identified. Such edges must be ignored. A status-array is normally used to keep track of such vertices.

Finally, Step 5 is added to identify strongly connected components. If *Low*[*i*] is equal to *StartTime*[*i*], then there is no back edge or cross edge that provides a path from Vertex *i* to its parent. Therefore, Vertex *i* is in a different strongly connected component than its parent.

5. If Low[i] is equal to StartTime[i] Identify a new SCC.

Tarjan's algorithm also includes a stacking mechanism to identify the vertices belonging to a particular strongly connected component, but because our aim is to create a single strongly connected component, we will not consider this mechanism further. The same is true for the mechanism that tags vertices that have already been assigned to a strongly connected component. The full Tarjan algorithm is given in Algorithm 3.

1. Function call DFS(i)
2. Set StartTime[i] to GlobalStartTime. // vertex i is first seen
3. Set Low[i] equal to StartTime[i]
4. Increment GlobalStartTime by 1.
5. For each vertex j adjacent to vertex i
   a. If StartTime[j] is equal to -1 Call DFS(j), Low[i] = min(Low[i],Low[j])
   b. Else if j is not already in a SCC, Low[i] = min(Low[i],StartTime[j])
6. If Low[i] equals StartTime[i] Identify a new SCC

Algorithm 3. The Full Tarjan Algorithm.

3 The modified Tarjan algorithm

Our primary modification to the Tarjan algorithm is in Step 6, which identifies new strongly connected components. We wish to prevent this detection from taking place. In Step 6, the algorithm is backing up from Vertex i. All vertices with start times greater than or equal to *StartTime*[i] are in the subtree rooted at Vertex i. All vertices, j, that have start times less than i must meet one of the following three conditions.

1. Vertex j is an ancestor of Vertex i in the DFS tree.
2. Vertex j has already been assigned to a strongly connected component.
3. Vertex j and Vertex i have a common ancestor k, and k is reachable from j.

Ignoring condition 2, it is clear that in Step 6, that Vertex i will be in the same strongly connected component as its parent if and only if there is an edge from a vertex v with a start time greater than or equal to *StartTime*[i] to a vertex w with a start time less than *StartTime*[i]. We modify Step 6 to insert such an edge when *Low*[i] is equal to *StartTime*[i]. Doing this also insures that condition 2 can never occur. Our modified algorithm is given in Algorithm 4.

1. Function call DFS(i)
2. Set StartTime[i] to GlobalStartTime. // vertex i is first seen
3. Set Low[i] equal to StartTime[i]
4. Increment GlobalStartTime by 1.
5. For each vertex j adjacent to vertex i
   a. If StartTime[j] is equal to -1 Call DFS(j), Low[i] = min(Low[i],Low[j])
   b. Low[i] = min(Low[i],StartTime[j])
6. If Low[i] equals StartTime[i]
   a. Set x to a random integer in the range [StartTime[i],GlobalStartTime]
   b. Set y to a random integer in the range [0,StartTime[i]-1]
   c. Identify the vertices v and w that correspond to the start times x and y.
   d. Add an edge from v to w.
   e. Set Low[i] equal to y.

Algorithm 4. The Modified Tarjan Algorithm.

Step 6c requires the inversion of the function that assigns start times to vertices. This is done in constant time by using an Inverse Start Time (IST) array and the following step following Step 2.

2.5 Set IST[StartTime[i]] to i

In Step 6c, v and w correspond to IST[x] and IST[y], respectively. Step 6e negates the *Low*[i] equals *StartTime*[i] condition, since Vertex i is now in the same strongly connected component as its parent. The addition of the new edge may make the *Low* value for some of the other vertices in the tree rooted at Vertex i incorrect, but since these values will not be accessed after backing up from Vertex i, they do not need to be changed.

Once Algorithm 4 has been run, the Gilbert or Erdős–Rényi models can be applied to add additional edges to the graph. The graphs created by Algorithms 1 and 4 are strongly connected and contain no more than $2n-2$ edges. Algorithm 1 adds $n-1$ edges. Algorithm 4 can add at most one edge per vertex, and cannot add an edge to Vertex 0. Algorithm 4 can add edges in any possible way to make the tree strongly connected. When combined with the Gilbert or Erdős–Rényi models any strongly connected graph can be generated. Both Algorithm 1 and Algorithm 4 are O(n). The Gilbert or Erdős–Rényi models are both worst-case O(n^2) because they must consider all $\frac{n^2-n}{2}$ potential edges.

Although Algorithms 1 and 4 can generate any strongly connected tree up to isomorphism, there are many isomorphs that will never be generated. For example, Algorithm 1 always adds an edge from Vertex 0 to Vertex 1. If this is a problem, it can be solved by a random relabeling of the vertices after the graph is generated.

4 Experimental Results

We ran several experiments to verify the effectiveness of our algorithm. The first experiment was to generate 10,000 5-vertex graphs, using the Gilbert model with $p=0$, to verify that the examples of Figures 1 and 2 could be generated. The algorithm generated both these examples, along with many others. Figure 3 contains a sample of the generated graphs.

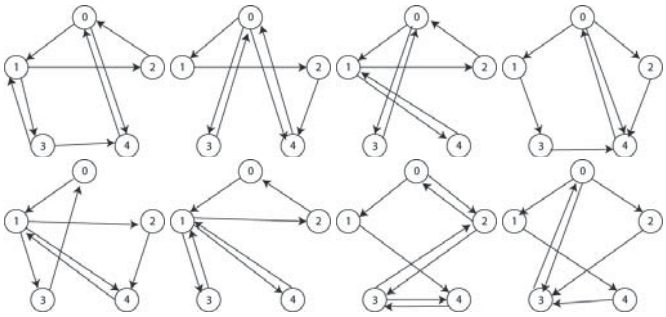

Figure 3. Some Sample Graphs.

Figure 3 demonstrates the essentially random nature of the generation process. Despite the fact that there is always an edge between Vertex 0 and Vertex 1, the structure of the graphs is obviously quite random, and they are obviously all strongly connected. This experiment was run using the Gilbert model with edge probability set to zero. This was done to show the structure of the strongly connected tree.

Four other experiments were run to determine the performance of the algorithm. The hardware was an Intel 3.40 Ghz core I7-2600 with 4 cores and 8GB of memory, running Linux Red Hat version 3.10. A

single core was used to run the experiments. The Linux *time* command was used to obtain the timings. Algorithm 4 was implemented iteratively rather than recursively for increased performance. The results of the experiments are given in Figure 4. The first experiment generated one million five-vertex graphs to show the performance of the algorithm with small graphs. Because the time to generate a small graph is tiny, it was necessary to generate a large number of graphs so that the execution time would be measurable.

The other three experiments were designed to show performance with large graphs. A single graph was generated because the time to generate such a graph is measurable. The Gilbert model was used for all four experiments (each edge having the same probability.) For the first experiment, we set the edge probability to .5, but it was necessary to set the edge probability to zero for the other three experiments due to memory requirements. If the edge probability were set to .5 for the one million vertex graph, half a terabyte of RAM (at least) would be required to store the graph.

Experiment	User time
1,000,000 5-vertex graphs Gilbert $p = .5$	3.8 seconds
One 1,000,000-vertex graph, Gilbert $p = 0$	.577 seconds
One 10,000,000-vertex graph Gilbert $p = 0$	7.3 seconds
One 100,000,000-vertex graph Gilbert $p = 0$	59.3 seconds

Figure 4. Experimental Results.

We speculate that it would take about 10 minutes to generate a one billion vertex graph, but 8GB of memory is insufficient to generate a graph of this size.

5 Conclusion

The algorithm presented here is simple, easy to implement, and very fast. It can generate any strongly connected graph when used in conjunction with the Gilbert or Erdős–Rényi models, and possible node-relabeling. The algorithm should prove to be a useful tool for the generation of strongly connected graphs in most contexts.

We have not yet addressed the vertex-oriented models, power-law and degree-sequence. Because each vertex has both an in-degree and an out-degree, it is not clear how to apply these models to directed graphs. It is necessary that the total of the in-degrees equal the total of the out-degrees. One model is to insist that the two degrees be identical for each vertex. Another model is to use the same set of degrees, but randomly distribute them over the vertices. It is also not clear whether the degree distributions should include the strongly connected tree edges, or whether these edges should be considered separately. For some degree distributions, it is not clear that a strongly connected graph even exists. We are currently working on these problems.

6 References

1. Maurer, P., "Generating Test Data with Enhanced Context-Free Grammars," *IEEE Software*, Vol. 7, No. 4, July 1990, pp. 50-55.
2. Maurer, P., "The Design and Implementation of a Grammar-Based Data Generator," *Software Practice and Experience*, Vol. 22, No. 3, March 1992, pp. 223-244.
3. Calvert, K., Doar, M., Zegura, E., "Modeling Internet topology," *IEEE Communications Magazine*, Vol. 35, No. 6, June 1997, pp. 160-163.
4. Gilbert, E., (1959). "Random Graphs" *Annals of Mathematical Statistics*. Vol. 30, No. 4 1959, pp. 1141–1144.
5. Erdős, P.; Rényi, A., "On Random Graphs. I," *Publicationes Mathematicae*, Vol. 6, 1959, pp. 290–297.
6. Aiello, W., Chung, F., Lu, L., "A Random Graph Model for Power Law Graphs," *Experimental Mathematics* Vol. 10, No. 1, 2001, pp. 53-66.
7. Chatterjee, S., Diaconis, P., Sly, A., "Random Graphs with a Given Degree Sequence," *The Annals of Applied Probability*, Vol. 21, No. 4, 2011, pp. 1400–1435.
8. Devroye, L, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.
9. Tarjan, R., "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, Vol. 1, No. 2, 1972, pp. 146–160,