A Random Graph Generator

Monica Van Horn
Department of Computer Science
University of Minnesota – Morris
vanh0087@umn.edu

Angela Richter
Department of Computer Science
University of Minnesota – Morris
rich0464@umn.edu

Dian Lopez (Advisor)
Department of Computer Science
University of Minnesota – Morris
lopezdr@mrs.umn.edu

Abstract

Random graph generation is commonly used in studying solutions to approximation algorithms. If random graphs can be generated, they provide a way to test algorithms for hard problems that have no optimal solution. By using these graphs, simulations can be used to determine, on the average, how well an algorithm performs. It is also possible to generate 'bad' graphs that will be able to approach some worst case solutions to algorithms. This paper will discuss the design and implementation of a random graph generator as well as its use on a scheduling problem for distributed/parallel systems. The design of our generator must take into consideration the number of nodes in the graph, the number of levels of the graph, the number of children for each node, and the height and width of each graph. Many truly random graphs turn out to look very similar to each other. That is why we also decided to generate graphs that were 'random' under certain constraints that would better test our algorithms. Lastly, we will discuss how the generator could be used to help others simulate their graph algorithm problems.

Introduction

Our research in random graph generation is part of a larger research project with individuals working on the NP-Hard problem of optimal scheduling in distributed computing. Distributed computing is a type of computing in which multiple workstations are utilized to simultaneously execute tasks that would be done sequentially on a single workstation. Therefore, a large portion of our research group is devoted to the study of how to efficiently schedule tasks on a group of workstations networked together.

Our part of the larger project was the development of a random graph generator to aid in testing the design and development of an approximation algorithm that deals with the NP-Hard problem of optimal scheduling of tasks in a distributed system. Our goal was to develop a random graph generator that would construct completely random graphs and that would also follow the constraints of the approximation problem.

Precedence Graphs

One constraint of our research is that we must generate precedence graphs with execution times. A precedence graph is a directed, acyclic graph where nodes represent sequential tasks and where arcs, for example, node i to node j, require that task i be completed before task j can start (See Figure 1). The arcs are weighted with weights representing the time it takes to communicate a result from one task to another if they are placed on different processors.

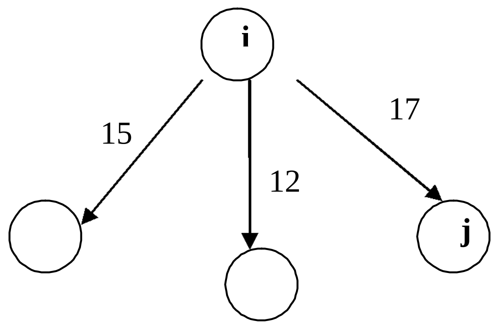


Figure 1. A Precedence Graph

The random graphs that our graph generator outputs must follow the constraints of the approximation problem. They must be acyclic, directed, and rooted. This is because in scheduling tasks in a parallel system, tasks need to be executed in sequential order, moving from one processor to the next. Cycles also need to be eliminated to ensure that the precedence constraints are met.

Precedence graphs are essentially spanning trees with some added edges. We will now discuss why we used spanning trees to aid in random graph generation.

Why Use Spanning Trees?

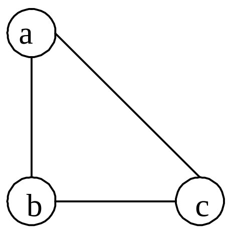
We must consider how to build a random graph that meets our constraints. We start with a complete graph and randomly weight the edges. The idea is to use a minimum spanning tree algorithm to generate a tree from the complete graph. A tree is already acyclic, so that constraint is met. A tree such as this can also easily be made into a directed graph with direction going from the root and proceeding downward through the levels to the leaves of the tree.

Now we have transformed our complete graph with random edge weights into a random precedence tree. Because our tree has weighted edges, we fulfill some of the data we need for our problem.

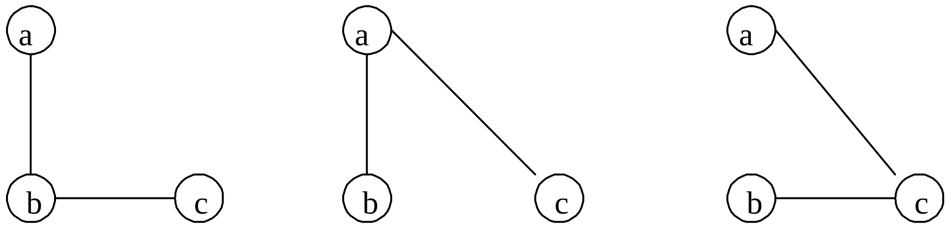
Only one issue remains to make this random tree fit the problem completely. We must add directed edges from random nodes to random nodes on lower levels to make the random precedence tree into a random precedence graph.

Random Precedence Graph Algorithm

A spanning tree of a graph is a subgraph that contains all of the vertices or nodes and forms a tree. We call it a spanning tree since it "spans" the graph. A graph may have many different spanning trees. For example, here is a graph of three vertices.



This graph has three different spanning trees:



A minimum spanning tree is defined to be a tree connecting all nodes that has the smallest total cost or weight, which is measured as the sum of the costs of the tree edges. Using Prim's algorithm[1], we grow a minimum spanning tree from one randomly chosen "root" vertex. As mentioned above, one of the constraints of the approximation algorithm is that graphs generated should have only one root. In each iteration of the algorithm, we choose the minimum-cost edge connecting a vertex, v, in the collection of vertices chosen so far to another vertex, u, outside of that collection. The vertex, u, is then brought into the collection and the process continues until a spanning tree is formed. Since the smallest-weighted edge is always chosen, it is guaranteed to always add a valid edge to the minimum spanning tree.

In order to keep track of insertions and deletions from the graph, we implemented a heap priority queue. The data structure of a heap allows these insertions and deletions to be achieved in logarithmic time. To accomplish this, the data structure stores only the elements and keys in a binary tree. All vertices are stored in the priority queue based on a key field. For each vertex, the key is the minimum weight of any edge connecting that vertex to another vertex already in the tree. The algorithm terminates when the priority queue is empty.

Prim's Algorithm

The following is the pseudo-code for Prim's algorithm of a minimum spanning tree that we used in our random graph generator.

```
Prim-MST(connected graph G, root r)
    1   for each u in the set V[G]
                    u is a vertex in G and V is the set of vertices of G
    2       do key[u]
                    key[u], the minimum weight of any edge connecting u to a vertex
                    in the tree, is set to infinity initially
    3           parent[u]    NIL
                    We begin with a set of vertices with no edges.
    4   key[r]    0
                    the root has no parent
    5   Q    V[G]
                    Initializes Q, a minimum-priority queue to contain vertices in G ordered
                    according to key[]  (edge weights).
    6   while Q   empty set
    7           do u    EXTRACT  -MIN(Q)
    8               for each v in the set Adj[u]
    9                   do  if v in the set Q and w(u, v) < key[v]
    10                          then   parent[v]      u ;      key[v]      w(u,v)
                     add v to the tree since it has the smallest edge weight so far
```

We now have a random tree with random edge weights. A precedence graph, however, can have more directed edges as long as they do not form a cycle. To form our precedence graph, we add zero or more directed edges to the tree, randomly choosing the edges from the initial complete graph. Our algorithm chooses an edge, then directs it according to the level of its nodes in the spanning tree. The edge is directed from the node at the highest level to the one at a lower level in the tree. If both nodes are at the same level, a random direction is chosen. If the added edge does not form a cycle, it is then made a permanent part of the precedence graph.

We started with a set of all possible edges and, using the Minimum Spanning Tree Algorithm, constructed a tree with random edges. By randomly adding edges to the tree, we formed a precedence graph. By repeating this algorithm many times, we can construct a set of random precedence graphs that can be used as a testbed of graphs on which to test other algorithms.

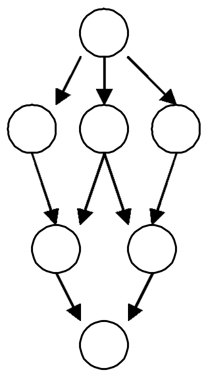
Using Random Graphs

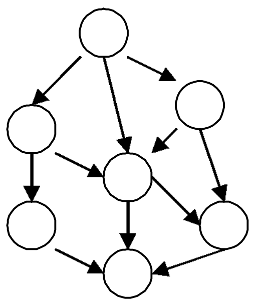
It is very useful to be able to use random graphs when testing an approximation algorithm. We need to be able to show that our algorithm can come close to the optimal solution. However, finding the optimal solution takes an exponential amount of time. By using completely random graphs, we can compute a set of optimal solutions using a brute force algorithm and then use the approximation algorithm on the same set of graphs. By comparing the results of the approximation algorithm to the optimal results, we can have an idea about how well our approximation algorithms works. If we did not use random graphs for this comparison, the output could be biased.

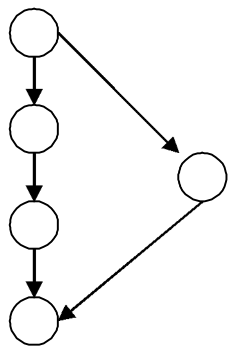
Results and Future Work

We have currently implemented the random tree generator. We are in the process of completing the precedence graph generator. We will then be able to generate a testbed of completely random graphs; graphs with a random number of nodes, a random number of levels, and a random number of children for each node.

Our next step will be to generate a testbed of graphs to use with approximation algorithms being developed in the area of scheduling tasks for a parallel/distributed system. For such problems, it sometimes becomes necessary to test the algorithms on certain types of graphs to see how well or, more often, how poorly they perform. With this in mind, we are planning to try to generate random graphs that exhibit certain properties. To do this randomly, we will generate a large number of graphs and then choose those that meet the criteria we are looking for. Examples of resulting graphs we will be able to display are shown below.


Cluster Structure


Highly Connected


Long Vertically

Conclusions

We have designed a method for constructing random precedence tree graphs for use in testing approximation algorithms. The addition of edges to the random minimum spanning tree may not produce completely random graphs. This is because of the exponential number of possible combinations to choose from in deciding which edge to add. The resulting almost random graphs, however, are very useful in providing a testbed for scientists to work with when testing performance of approximation algorithms.

Acknowledgements

This work was sponsored by: Collaborative Research Experience for Women (CREW). Thank you to Joan Francioni, CREW Program Director and to the rest of the committee.

References

[1] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2001). *Introduction to Algorithms* (2nd ed). MIT Press and McGraw-Hill.

[2] Weiss, Mark Allen (1998). *Data Structures & Problem Solving Using Java*. Addison Wesley Longman, Inc.

[3] Goodrich and Tamassia (2001). *Data Structures and Algorithms in Java* (2nd ed). John Wiley & Sons, Inc.

[4] Lopez, D. R., O'Brien, D., Krohn, J., Nelson, J., Wilfahrt, R. (2002). *Design and Testing of Parallel/Distributed Precedence Tree Scheduling Algorithms.* Midwest Instruction and Computing Symposium.