Probability of Connected Graphs

An investigation of graph connectivity

Qingbo Liu - May 8, 2018

Introduction

In this project we explore the probability of a graph being connected by generating random graphs and test their connectivity. We use the random generator provided by Java's library, which produces pseudo-random numbers. Both directed and undirected graphs are produced and tested for their connectivity. The data shows that undirected graphs have a higher chance of being connected, which is reasonable given that undirected graphs allow edges to be connected in two directions.

Methodology

I. Random Number Generator

The Random class provided by Java's util library is used for random number generation. Because of the deterministic nature of the algorithm implemented in Random class, two Random objects fed with identical seeds will always yield the same sequence of random numbers. Though the algorithm is well designed to yield random numbers that are uniformly distributive over a given range, this still causes problems for the project because the project aims for truly random graphs. To compensate for this, the time when a new Random object is initialized is fed into the object as the seed, expressed in epoch time format, as an attempt to alleviate the issue of randomness.

As an aside, the project has also tried True Random Number Generators (TRNGs), as opposed to Pseudo-Random Number Generators (PRNGs) implemented by Java's library, by using API provided by random.org, which generates random numbers based on atmospheric noise. However, because the API is still in beta version, a limit of a thousand requests per day is imposed and therefore cannot satisfy the amount of data required for this project.

II. Random Graphs Generation and Connectivity Detection

Random graph generation is based on random number generator. The graph uses adjacency matrix to represent edges between vertices.

For undirected graph, breath-first search is used to detect connectivity, while Tarjan's algorithm, which is based on the idea of depth-first search and stack, is implemented for directed graphs.

Data Analysis

I. Data

To compensate for the huge fluctuation of random numbers, for each combination of graph size and edge size 1000*1000 graphs are generated and tested for connectivity.

The table below shows the percentage of graphs generated that are connected. Note that the edge size here denotes the range of random numbers that a random number generator could produce, e.g. 1000 refers to the range [0..1000]. The graph size is fixed at 1000, a reasonably large number.

Graph Kind/ Edge Size	1000	2000	5000	10000
Undirected	7.22%	9.94%	15.30%	21.77%
Directed	8.05%	9.89%	14.49%	20.45%

Figure 1. Probability of connectivity for undirected and directed graphs

II. Discussion

When the edge size is 1000, the probability of connectivity of directed graphs is surprisingly higher than that of undirected graphs. This suggests that it might be easier to form a connected graph for directed graphs than undirected graphs when the edge size and graph size are the same. But as the edge size is growing, the gap of probability between directed and undirected graphs starts to widen, with undirected graphs on the winner side. This result follows the intuition about the graph size.

Time Complexity

The process of producing random numbers is $O(1)$, according to the documentation provided by Oracle. The overall time complexity then depends on the number of graphs generated in each run and the size of each graph. Because of the probabilistic nature of the generative process, only worst-case time complexity is taken into account.

The time complexity of the overall graph generation is $O(nm)$, where n is the number of graphs generated and m is the upper bound of the graph size. The process of producing a single graph is $O(m)$, because the number of edges is just a multiple of the graph size and

therefore filling edges between vertices takes $O(m)$. There are in total n graphs produced, so the overall running time is $O(nm)$.

Implementation

I. Parallelism

Because of the large time complexity of the algorithm, it takes several hours to finish the generative process in a single thread environment. To speed up the running time as much as possible, the project implements a thread manager and a Runnable class to divide the task of generating random graphs. The thread manager is responsible for both firing new threads and collecting data. When a thread has finished processing, the thread will pass the result to thread manager, which is passed as a parameter to the constructor of threads, an implementation of the callback mechanism.

Note that this implementation has been removed in the final version, but is kept in the git log, specifically the commit with the message "everything works right now, albeit very slow".

II. Design Patterns

Besides the parallel implementation, this project also adopts the command pattern, where an abstract parent class specifies the abstract methods that concrete subclasses must implement. The parent class then takes advantage of the abstract methods in its invariant implementation, without concern about the part of implementation that must be changed for each subclass.

The classes that implement command pattern are NumberGenerator and Graph, with their subclasses filling the holes specified in the parent classes.

References

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Random.html
[2] https://www.random.org/randomness/
[3] https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/

[4] https://www.youtube.com/watch?v=TyWtx7q2D7Y&t=826s