CS 498A
Undergraduate Project Report

Distributed Graph Algorithms And Generalized Architecture For Some Graph Problems

Submitted by

Abhilash Kumar 12014
Saurav Kumar 12641

Under the guidance of
Dr. Arnab Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology , Kanpur
Uttar Pradesh, India – 208016

Abstract

The number of connected sub-graphs of a given graph can be exponential in number of vertices. The significant growth of graph data in several applications creates a need to build systems and algorithms which utilize the immense potential of distributed computing. In this project, we have designed a generic distributed architecture to solve the problem of computing all connected sub-graphs of a given graph. We will also show how to use the architecture to solve similar graph problems like finding all spanning trees or finding all maximal cliques of a given graph.

Contents

Chapter 1

Problem Statement

The problem consists of 2 parts:

- Designing an algorithm to compute all connected sub-graphs of a given graph, in a distributed environment.
- Designing a generic distributed architecture to solve the above problem and proposing how to solve other similar problems using the same architecture.

Chapter 2

Introduction

We will focus on the problem of generating all connected sub-graphs of a given graph. First, we will design an algorithm to solve this problem. Then we will describe a distributed architecture using which we can solve the problem efficiently in a distributed environment. Later, we will show how we can design similar algorithms to compute all spanning trees, all paths, all cliques etc using the same architecture.

So, our first problem is to compute all connected sub-grpahs of a given graph. Since the number of connected sub-graphs can be exponential[1], our worst case complexity will anyway be exponential. We will be looking for ways to use distributed computing to solve this problem in scalable manner, i.e time taken to compute all the connected sub-graphs is inversely proportional to the number of servers used for computation.

We will also provide an implementation of the algorithm and the architecture in Python and present various simulation results. We will also describe a few optimizations that we did in our architecture to reduce the practical running time.

Chapter 3

Algorithm and Architecture

Connected sub-graphs exhibit *sub-structure*, i.e each connected sub-graph is made up of one or more smaller connected sub-graphs. Given a smaller sub-graph, we can extend it by adding exactly one outgoing[1] edge to generate a larger sub-graph, and we can enumerate all connected sub-graphs by just repeating this process over and over again, given a set of base case graphs. Base cases are sub-graphs represented by all the edges of the graph, i.e graphs consisting of exactly 2 nodes with a single edge between them.

3.1 Algorithm

As stated above, we intend to create 2 functions: *initialize* and *process*. The *initialize* function creates the base case sub-graphs, and *process* function extends a given sub-graph as follows.

- **Initialize**:
  - Queue Q
  - For each edge in G
    - Create a sub-graph G' representing the edge
    - Push G' to Q

[1] An extensible edge, present in the given graph but not present in current sub-graph

- **Process**:
  - while Q is not empty
    - G = Q.pop()
    - Save G
    - For each outgoing edge E of G
      G' = G ∪ E
      if G' has not been seen yet
      Push G' to Q

Example

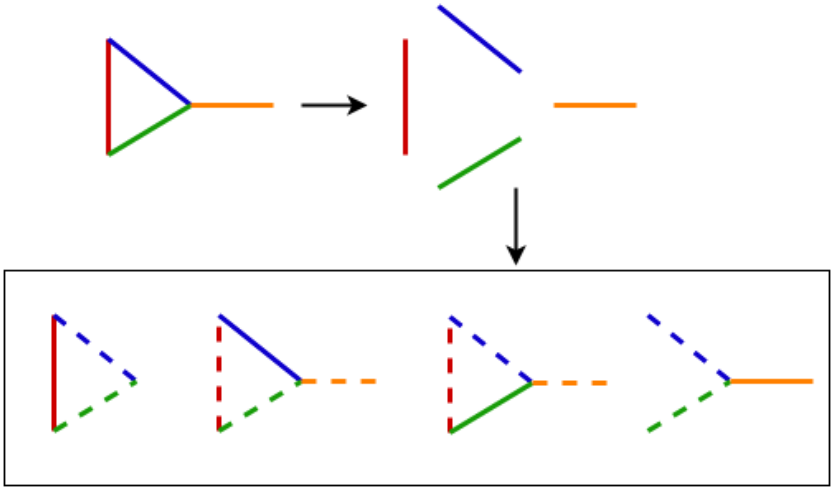

Figure 3.1: Generating initial sub-graphs from a given graph

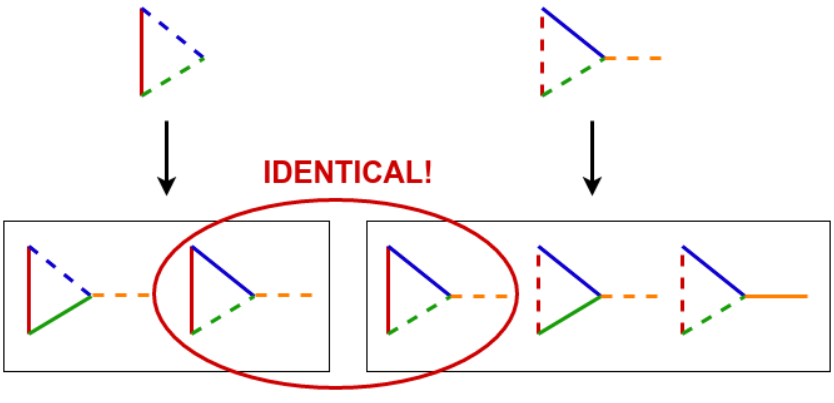

Figure 3.2: Extending a sub-graph to generate new sub-graphs

3.2 Architecture

Let us now describe the distributed architecture that we are going to use to solve the above problem. The architecture is inspired from Hadoop's[2] Master-Slave architecture. Such master-slave framework is frequently used in parallel and distributed applications.

3.2.1 Master Slave architecture

We have a set of servers with one master server. The main job of the master server is to assign initial tasks to slaves, keeping track of live slaves, collecting result and other bookkeeping jobs. All servers communicate through message passing over TCP.

Master assigns initial *tasks* to the slaves and finally may or may not collect the final or intermediate results. Assigning task to a slave is done by message passing between the master and the slave and similarly, the final or intermediate result of computation may be returned.

A *Task* object represents a sub-graph and it contains all the necessary information that may be required to process the sub-graph. Each slave has its task queue from where it picks up tasks to process. A slave may request a task from other slaves when its task queue is empty. Clearly, processing ends when task queues of each slave are empty.

3.2.2 Tasks, Task Queue and Bloom filter

Each **task** denotes a unique connected sub-graph which has been computed. It contains a list of vertices that are present in this sub-graph and a list of edges that can be used to extend this sub-graph. Thus, *processing* a task generates zero or more new tasks, which is added to the task queue of the processing slave. An idle slave may requests other slaves for tasks, which then pops a task and returns it to the caller. This ensures a fair division of tasks among slaves.

Each slave picks up a task from its task queue and processes(extends) it. Unique tasks among the newly generated tasks are pushed back into the task queue. While extending, it must be ensured that a previously processed task is not pushed back in the task queue. In order to ensure uniqueness, the challenge is that the task may have been processed on another slave. To solve this, we maintain a global distributed *bloom filter* to check the uniqueness of a generated task.

Bloom filter[2] is distributed over the slaves so that none of the servers get loaded with uniqueness checking requests. Also, each bloom filter is responsible to handle requests for hash values of a certain range only and this range is pre-determined. All the slaves have this information prior to the start of processing. This ensures that no task gets processed twice and the architecture remains fairly scalable, as on increasing the number of slaves, each slave will do lesser amount of processing and each slave will be responsible for smaller range of hash values.

3.2.3 Why bloom filter over hashing?

We use bloom filter[4] because it is very space efficient as compared to hash tables. Space required to get error probability[3] of p is

$$\frac{-n \times \ln p}{(\ln 2)^2} \ bits.$$

Though bloom filter is non-deterministic but its error probability can be reduced to negligible values with very little extra space. Deterministic hash tables or search trees can be used to make the algorithm deterministic but it will greatly increase the space required(RAM) and running time. Another huge advantage of bloom filter is that it can also be parallelized whereas

[2]A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set.

hashing cannot be. A parallelized bloom filter is nearly $\mathcal{O}(1)$.

3.2.4 Using the architecture

So far we have described our algorithm to compute all connected sub-graphs and a distributed architecture which processes some tasks in scalable manner given that processing of each task produces zero or more new tasks, and tasks are hashable objects. Now we will describe how to combine these two to produce a scalable solution to our problem.

In order to use the architecture, the tasks must follow the property that processing a task produces zero or more new tasks and processing ends when no task is left to process. The architecture ensures that a task is processed exactly once, using the distributed bloom filter.

As stated earlier, the architecture requires only two functions: *initialize* and *process*. The function **initialize** generates initial tasks and the master server randomly assigns these tasks to the slaves. The second function **process** defines a procedure that will generate new tasks from a given task (extend sub-graph in our case).

3.2.5 Fitting connected sub-graph problem

So, in the problem that we want to solve, **initialize** will create base tasks: one for each edge of the given graph and the master will distribute these tasks randomly among the slaves. The function **process** will take a connected sub-graph and extend it by adding all extend-able edges, one at a time. For each of the larger sub-graph it will update the list of edges on which they can be extended, forming new tasks.

Chapter 4

Simulation and Results

4.1 Simulation

4.1.1 Machine configuration and test cases

In order to test and benchmark the architecture, we simulated the procedure using 2 machines, say H and L, with following configurations:

- H: 24 core, 200 GB, Xeon E5645 @ 2.40GHz
- L: 4 core, 8 GB, i5-3230M CPU @ 2.60GHz

We opened multiple ports (6 on H, 2 on L) to mimic 8 slave servers. Using various combinations of them, we observed the performance of the architecture on a few graphs. We will present our findings about two graphs:

- G(14, 13): A tree with 14 vertices, such that one vertex (say 0) is connected to all other vertices.
- G(16, 15): A similar tree graph, with 2 more vertices and edges.

We chose to test with these two graphs as it is simple (manually) to compute number of connected sub-graphs in these cases. Clearly, G(14, 13) has 8191 and G(16, 15) has 32767 connected sub-graphs[1].

[1]In such a graph, picking any non-empty subset of edges makes it a connected sub-graph, so G(n+1, n) has $2^n - 1$ connected sub-graphs

4.1.2 Performance measures

We will present observations of following performance measures:

- Number of tasks processed by each slave for various combinations of slaves for G(14, 13) and G(16,15)
- Total and average number of hash check queries made by the slaves for G(14, 13) and G(16, 15)
- Total running time vs Number of tasks for various combinations of slaves

4.2 Results

4.2.1 For G(14, 13)

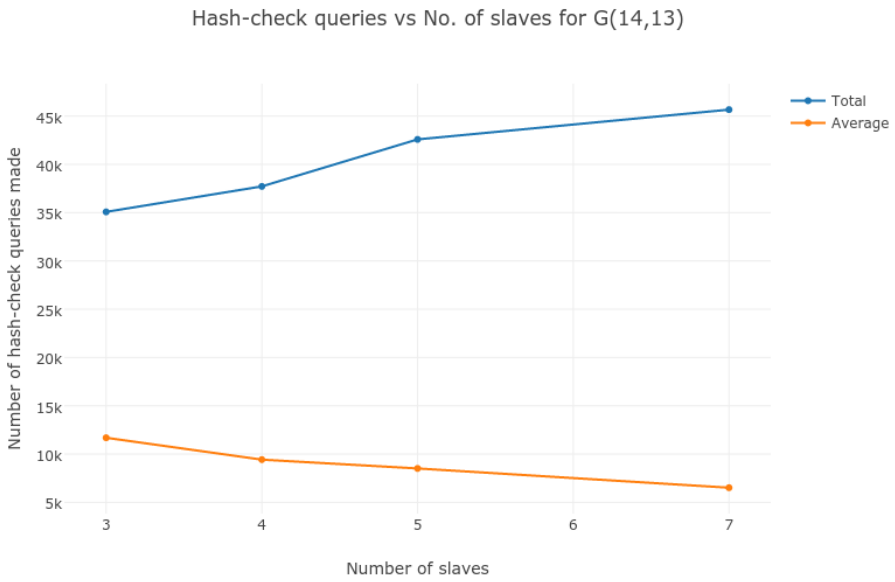

Figure 4.1: Number of hash check queries vs number of slaves for G(14, 13)

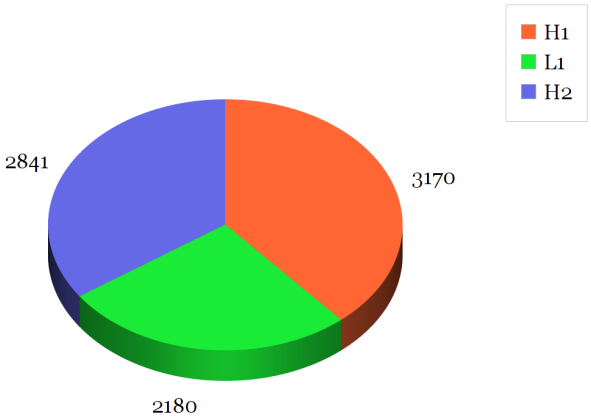

Figure 4.2: Distribution of number of tasks processed by slaves for G(14, 13)

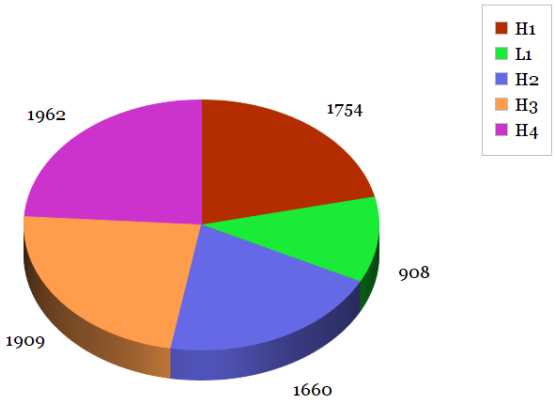

Figure 4.3: Distribution of number of tasks processed by slaves for G(14, 13)

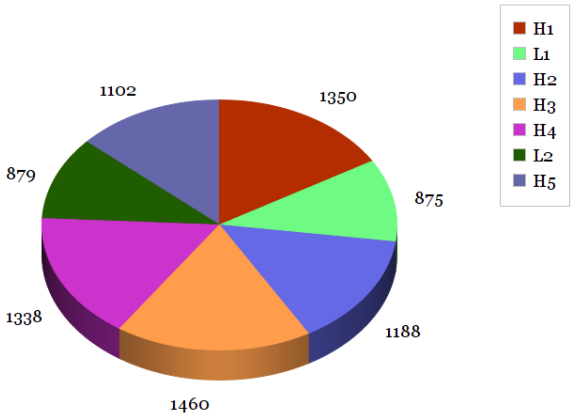

Figure 4.4: Distribution of number of tasks processed by slaves for G(14, 13)

4.2.2 For G(16, 15)

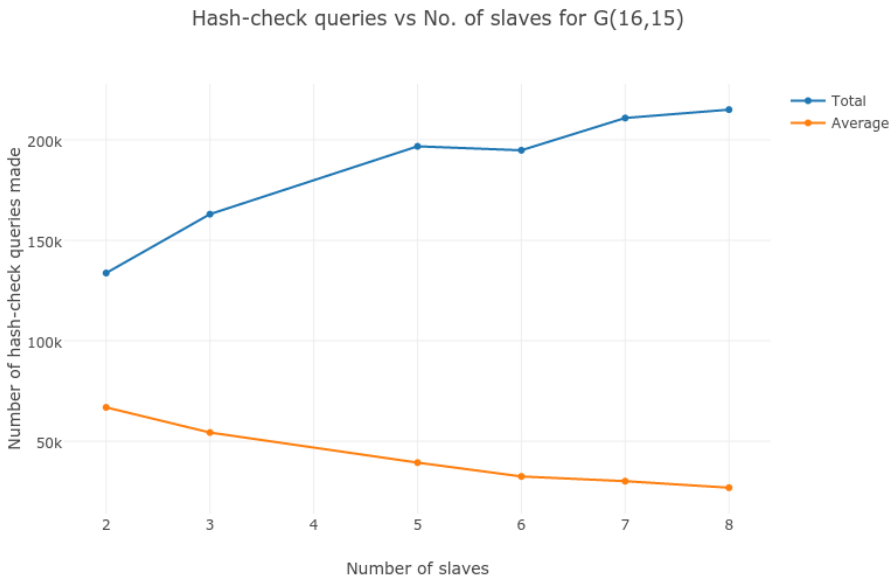

Figure 4.5: Number of hash check queries vs number of slaves for G(16, 15)

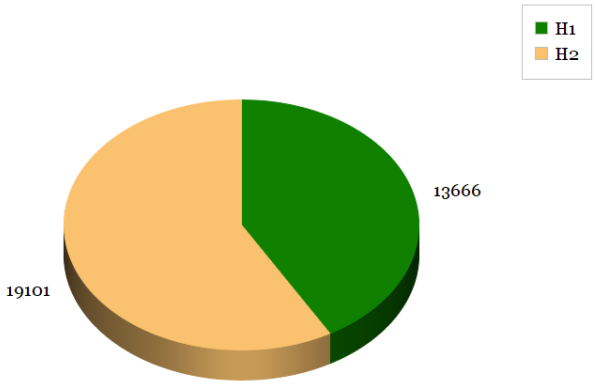

Figure 4.6: Distribution of number of tasks processed by slaves for G(16, 15)

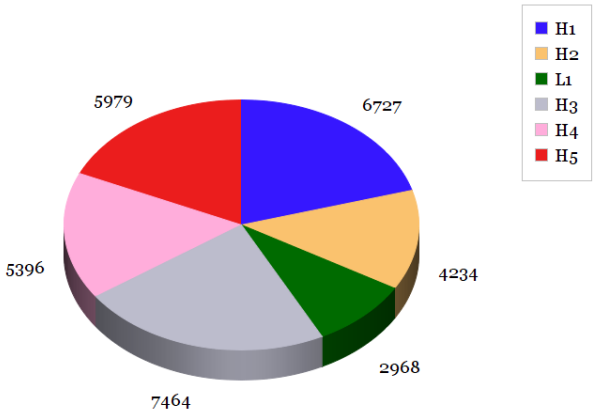

Figure 4.7: Distribution of number of tasks processed by slaves for G(16, 15)

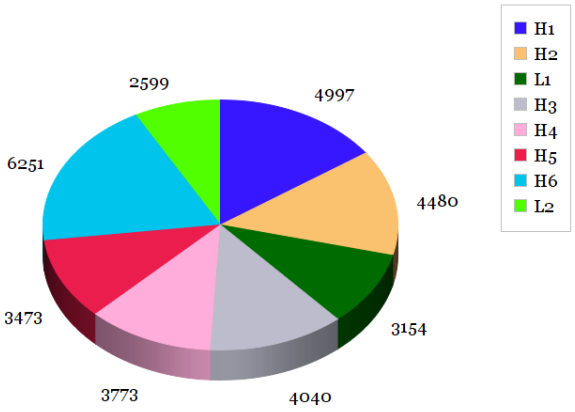

Figure 4.8: Distribution of number of tasks processed by slaves for G(16, 15)

4.2.3 Running Time

The architecture also supports associated computations per task. Assuming each such task takes 10 milliseconds, following is observed for 3 different graphs.

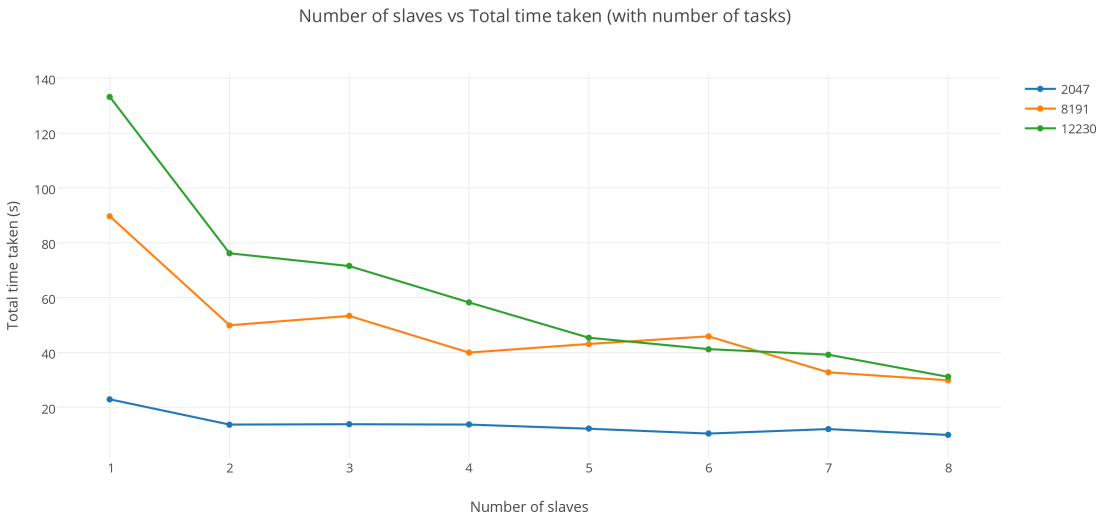

Figure 4.9: Running time when **process** does addition computation(10ms)

Chapter 5

Analysis and Conclusion

In this report, we have described a scalable solution to find all connected sub-graphs of a given graph.

5.1 Analysis

Let us analyze the efficiency of the algorithm. Let us denote the number of edges in the graph by m, the number of nodes by n and the number of connected sub-graphs by p. Let t denote the sum of number of edges in the graph all of these p connected sub-graphs. Also, let h be the total number of graphs for which we call hash check function. Clearly, any optimal solution cannot be better than $\mathcal{O}(t)$. The hash function that we have used, is based on the edges in the graph. Hence, hash of an extended graph can be calculated in $\mathcal{O}(1)$, given the hash of its parent graph. So, overall complexity of our solution is $\mathcal{O}(h * n + c * t)$, where c is the maximum number of times hash check is called for a particular connected sub-graph. Therefore, h is also bounded by $c * p$. Ideally, if this distributed solution is completely scalable, the overall complexity of the solution becomes $\mathcal{O}(c*(p*n+t)/k)$, where k is the number of slave servers used. It is important to note that $p*n+t$ is the lower bound on the optimal solution for this problem(in a non-distributed environment). So, our solution is better than *any* simple iterative solution if $c < k$.

Now, let us try to find lower bound on c. m is an obvious lower bound on c as hash check for any sub-graph can be called maximum for number of edges in that graph. Since the total number of edges in the graph is m, any sub-graph cannot have more than m edges. Suppose, we have a connected sub-graph with e edges then that particular sub-graph will itself have at least

2^e sub-graphs on its own. But, the total number of sub-graphs are t, so we conclude $c < min(m, log(t))$.

5.2 Advantages

The algorithm and architecture are highly scalable, i.e adding more slaves reduces total running time. More slaves can be added easily in this architecture. Tasks are evenly distributed among the slaves. The architecture itself takes care of things like efficient machines process more tasks, ensuring no slave is idle while others are processing. The architecture is highly reusable, i.e many other problems can be solved using this architecture. One needs to provide only 2 functions, **initialize** and **process**. The implementation is also network fault tolerant, i.e it handles network failures gracefully.

5.3 Solving other related problems

It is not hard to see this solution and architecture can be used to solve several similar problems too. Specifically problems which involve finding all sub-graphs of a graph with certain property that exhibit substructure property, i.e larger graph with the property that it contains smaller sub-graphs following the same property. Given a graph following this property, we can check whether its extended graph follows that property or not. We can solve problems like generating all cliques, paths, cycles, sub-trees, spanning sub-trees of a given graph. We can also use this architecture to solve few classical NP problems like finding all maximal cliques in scalable manner.

Chapter 6

Future Work

We would like suggest a few improvements in the architecture that may be done in the future.

First, the hash check requests to the bloom filter, being network calls, dominate the overall run time. Parallelizing the bloom filter will substantially decrease the run time.

Second, parallel execution of tasks on multi-core slaves. Since the tasks are processed independent of each other, they can be run in parallel threads on a multi-core machine for faster processing.

Third, the current implementation assumes that servers do not go down. The architecture recovers from minor network failures, but not when the server itself goes down.

Finally, handling storage when the task queue may get extremely large for a large input graph.

Acknowledgments

We take this opportunity to express our profound gratitude and deep regards to Dr. Arnab Bhattacharya for his exemplary guidance, monitoring and constant encouragement throughout the course of this project. The blessing, help and guidance given by him time to time shall carry us a long way in the journey of life on which we are about to embark.

Our thanks and appreciations also go to our friends who helped us during the course the project and people who have willingly helped us out with their abilities.

We would like to thank Python Software Foundation and open source communities for the software libraries which simplified our implementation.

Abhilash Kumar
Saurav Kumar

10 November 2015
Indian Institute of Technology Kanpur

Bibliography

[1] Welsh, Dominic (1997), "Approximate Counting", Surveys in Combinatorics, Bailey, R. A., ed., Cambridge University Press, pp. 287-324.

[2] Borthakur, Dhruba. "The hadoop distributed file system: Architecture and design." Hadoop Project Website 11.2007 (2007): 21.

[3] Mitzenmacher, Michael, and Eli Upfal. Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge University Press, 2005.

[4] Starobinski, David, Ari Trachtenberg, and Sachin Agarwal. "Efficient PDA synchronization." Mobile Computing, IEEE Transactions on 2.1 (2003): 40-51.

Resources

- Presentation:
  `http://www.slideshare.net/SauravKumar145/distributed-graph-algorithms`
- Source Code: `https://github.com/abhilak/DGA`