Graph Connectivity of Dynamically Connected Graphs

Akash Desai

Department of Computer Science

akashpra@buffalo.edu

University at Buffalo

1. Abstract:

In today's world of social networks, GPS and internet, everything is connected and they are continuously changing their dynamics with each passing moment. To represent these changes in connectivity of different elements, we take help of graphs. To understand the effect of these changes, we use graph connectivity to establish new relations and discard the previous ones.

Social platforms such as Facebook, Google+, and LinkedIn have already entertained this aspect in their newly added functionality. *"People you may know"* feature of Facebook is dependable on these newly established relations. On such big platforms, there are millions of users. Every such user is node in a graph and that shows us that size of the problem is huge. Graph connectivity of billion nodes would take days to calculate on a sequential machine and with every new connection you might have to re-calculate everything.

With such social platforms, time is not a critical issue while displaying the effects of dynamism. Suggesting a friend or a place to visit or page or professional connection can take days and nobody would notice or complain. So, why even bother with the computation time of such graph connectivity. Apart from social networks, this feature is also important for maps and GPS location. Now, these things need result fast and accurate.

This report is an application of the algorithm suggested by Robert McColl, Oded Green, David A. Bader and parallel programming (GPU based) to make system work faster.

Here, in this paper we have implemented a system which will determine that which computations are required, given there are thousands of request coming every second but not each and every one of them require systems attention. Some of them are redundant and we will figure that out to reduce computation cycles by using this algorithm.

We will explain how to handle this problem with help of this method.

2. Introduction:

As we have stated previously, computation time to check connectivity between any nodes is critical when it comes to map. There are other applications of this idea but for now we have identified challenges with regard of this problem.

On a global map of roads, while stating driving direction, we have to give user the shortest or least time consuming route. Providing leasing time consuming route takes lots of factor into consideration. Road blockage for some repairing task, due to excessive traffic, or due to some special occasion.

Let's address a simpler and more popular problem first: Finding a connected component, which is first step to getting to our goal.

Given any undirected graph (In our problem statement graphs would be undirected so it would be better to set the standards straight from the beginning.) G= (V, E), where V represents vertex set and E represents Edge set, connected component would be $C \subseteq V$ such that every vertex of C is connected to every other vertex. The graph can be deemed as connected graph if there is only one such component.

Finding the connected component is widely discussed and well researched problem. Hopcroft and Tarjan [2] presented one of the first approaches for partitioning a graph into connected components using a series of Depth First Searches (DFS), one for each component. Breadth First Search (BFS) can also be used in place of DFS. Approaches relying only on DFS and BFS are aimed at static graphs which can be thought of as snapshots of a dynamic graph at an exact time.

In maps, locations are vertices and roads that connect vertices are edges. Now, the rate of new edge insertion or deletion is very high. For the massive graphs, it is impractical to use static snapshots to get around the dynamic graph since they take long time to calculate.

To draw the analogy between insertion and deletion with maps, consider deletions as road blockage and insertion as reinstating of those previously deleted edges and new roads. This problem inspired researchers to find better way to do such calculation. From then to now there has been so many proposed solution.

In a massive graphs, while doing first computation using BFS (Breadth First Search) we allocate level and component id to each vertex to associate them with different component. This data is useful when new insertion in graph takes place. If we know the component id of associated vertices with newly inserted edge, we can do insertion operation in O (1) time. If component id of vertices associated are same, we simply add the edge and give new level detail to this vertices accordingly. But when the component ids are different then, insertion will join two different component into one single component in such case we have to reallocate same component id to both of them and change the level details.

In this case, new levels would be determined by size of both component id. We will give component id of larger component to the smaller one and new levels= previous level + inserted edge's larger component vertex's level+1.

In this work, we show how to maintain an exact labeling of the connected components of a dynamic graph with millions of edges while applying batches of edge insertions and deletions in parallel [1]. To accomplish this task, we employ a "parent-neighbor" sub-graph structure of up to a fixed size O (V) [1]. In this sub-graph, parent and neighbor relationships represent paths to the root of a breadth-first traversal of each component [1]. As long as each vertex has a path to the root, the component is unbroken [1]. In practice, we show that the average case for maintaining this approach is much faster than performing the O (V + E) work required to re-compute from scratch. The storage complexity is O (V) [1].

From here, we will talk about the linear implementation history and will discuss some brute approaches, from there we will give brief idea about current existing algorithms and will discuss our algorithm at length. After that we will go through implementation and results of my work and will discuss outputs at length.

3. Linear Implementation History:

In this section we will discuss about linear approaches used for solving this problem. This will include brute-force method (Intuitive method), and some other approaches.

1. **Intuitive Method:** So, given a graph G= (V, E) at first we need to label the each vertex, assign them a level accordingly. For that we will run BFS algorithm in the beginning to initialize these values. First run of BFS on graph G is initial static snapshot of default orientation. Now, each vertex has level and component id to identify itself with respective component. As we described earlier, most intuitive algorithm would be taking snapshot at given time continuously which can give us illusion of dynamic computation. For doing that, we have to run BFS each time new edge insertion or deletion takes place in the graph. This approach is impractical for massive graphs and we here are discussing massive graphs only.
2. **Union-find:** Our implementation is advanced work of this algorithm, so for understanding that you need to understand this algorithm. In this algorithm, after first BFS run, we collect information of different component and put vertices with same component id in same set. By doing that, when we get inquiry about the two points being connected or not, we only need to check this sets. If both the vertices are in the same set then they are connected. This approach will take O (V) time. In advanced methods, we will try to decrease this time significantly.

After first iteration, when new insertion come, check the component id of both vertices and if they are different then do the union of those 2 different set in one. We need to take care of levels though, but we will discuss that later. This method was useful in case of insertion only, deletions are unaddressed.

There are various other approaches that are faster than these two, but these are the most discussed methods and were the base of many algorithms came in later in time. Linear version of our implementation is about keeping information about each vertex's neighbor and their levels. We will discuss that in implementation part at length and we will take those results to compare with our parallel algorithm.

For now, we will rest our discussion of linear implementation to this only.

4. Other approaches:

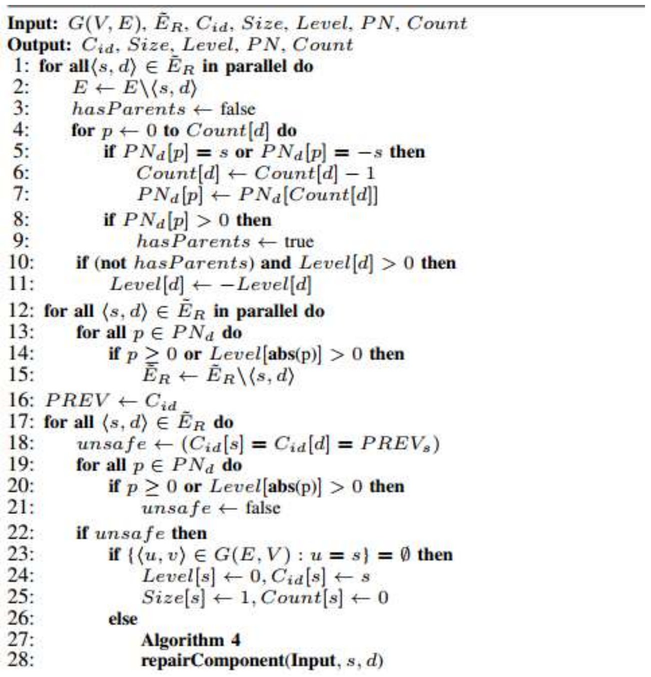
Along with our implementation, there are other very well-known algorithms. Here are the brief idea about some of them.

Shiloach and Even's algorithm keeps information about still connected component after each edge deletion. To achieve this they store information about levels of each vertex, so if on edge deletion both vertices are in the same level, connected component would be same as before, in other case vertex with higher level will look for neighbor with lower level value. If there is no such neighbor, component would break apart into 2 pieces. Current research shows that a sequence of graphs are maintained, one per each edge inserted, and reachability trees. In this case update time is O (E+ V Log V) and query time is O (V).This implementation laid base for the algorithm suggested by Robert McColl, on which our implementation is based.

In [2], presented technique helps us distributing dense graph into sparse graphs, by dividing the original graph into smaller sub-graphs, then they check connectivity for smaller graphs and then eventually merge the results.

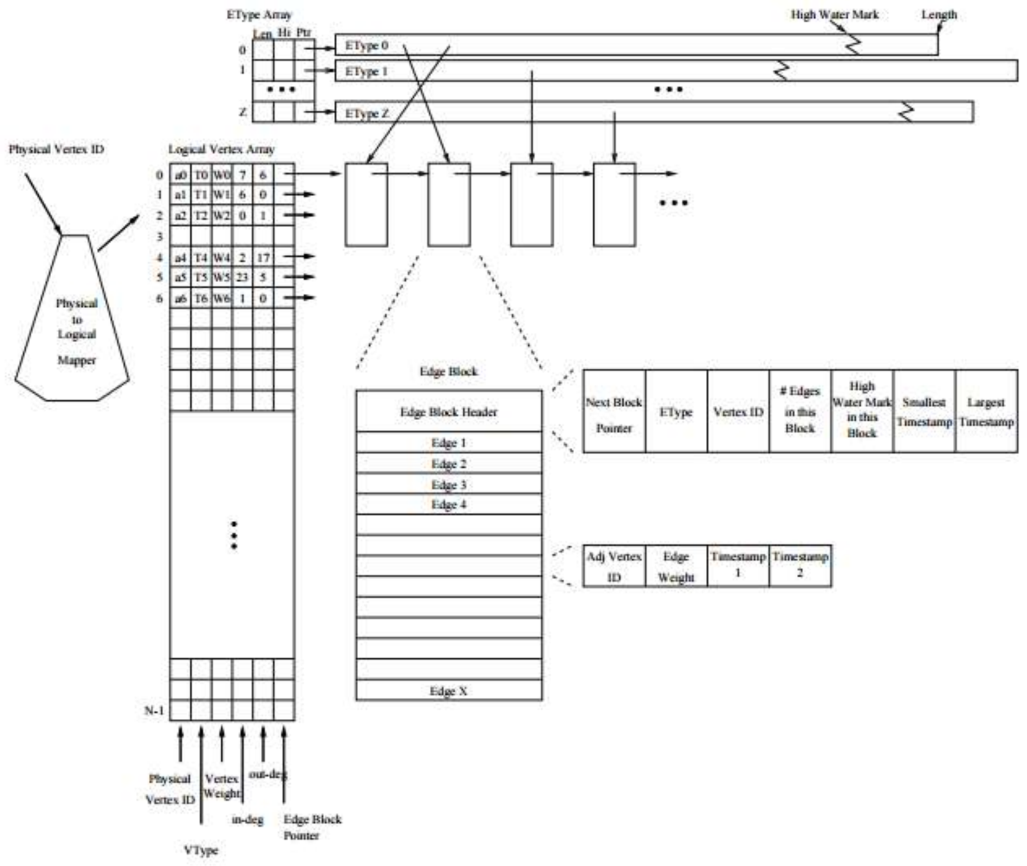
STINGER is a high-performance data structure designed for dynamic graph problems [1]. STINGER is a compromise between the massive storage and fast updates of adjacency matrices and the minimal storage and static nature of Compressed Sparse Row (CSR) representation [1]. STINGER has faster insertions and better locality than traditional dynamic sparse structures like adjacency lists [1]. Further, STINGER is designed for parallelism such that multiple threads can read and update the graph concurrently [1].This is our algorithm down below and under that we have shown STINGER data structure.

1. Our algorithm [1]. For more details refer to [1].



```
Input: $G(V,E)$, $\tilde{E}_R$, $C_{id}$, $Size$, $Level$, $PN$, $Count$
Output: $C_{id}$, $Size$, $Level$, $PN$, $Count$
1:  for all $\langle s,d \rangle \in \tilde{E}_R$ in parallel do
2:      $E \leftarrow E \backslash \langle s,d \rangle$
3:      $hasParents \leftarrow$ false
4:      for $p \leftarrow 0$ to $Count[d]$ do
5:          if $PN_d[p] = s$ or $PN_d[p] = -s$ then
6:              $Count[d] \leftarrow Count[d] - 1$
7:              $PN_d[p] \leftarrow PN_d[Count[d]]$
8:          if $PN_d[p] > 0$ then
9:              $hasParents \leftarrow$ true
10:     if (not $hasParents$) and $Level[d] > 0$ then
11:         $Level[d] \leftarrow -Level[d]$
12: for all $\langle s,d \rangle \in \tilde{E}_R$ in parallel do
13:     for all $p \in PN_d$ do
14:         if $p \geq 0$ or $Level$[abs(p)] $> 0$ then
15:             $\tilde{E}_R \leftarrow \tilde{E}_R \backslash \langle s,d \rangle$
16: $PREV \leftarrow C_{id}$
17: for all $\langle s,d \rangle \in \tilde{E}_R$ do
18:     $unsafe \leftarrow (C_{id}[s] = C_{id}[d] = PREV_s)$
19:     for all $p \in PN_d$ do
20:         if $p \geq 0$ or $Level$[abs(p)] $> 0$ then
21:             $unsafe \leftarrow$ false
22:     if $unsafe$ then
23:         if $\{\langle u,v \rangle \in G(E,V) : u = s\} = \emptyset$ then
24:             $Level[s] \leftarrow 0, C_{id}[s] \leftarrow s$
25:             $Size[s] \leftarrow 1, Count[s] \leftarrow 0$
26:         else
27:             Algorithm 4
28:             repairComponent(Input, $s$, $d$)
```

2.STINGER DATA STRUCTURE [3]



5. Implementation:

My implementation is extracted from Shiloach and Even’s algorithm with the mix of Robert McColl’s algorithm [1]. (Codes are attached with the zip file).

Here, we have maintained data structures who keeps information about the neighbors of every vertex. In which we store information about which vertices are neighbor to given vertex and what are their current level after latest run of BFS. In [1] we are suggested to keep information in PN data structure which puts limit on number of neighbor’s information that on can store but in our case, we kept a global structure accessible to every vertex. So, we don’t have to put an upper limit on number of neighbor’s information.

After initial run of BFS on given graph G= (V, E) we kept information about the levels in an array. Now, here we have two different test results, one where number of requests are 500 and other has 1000. So we store all the request in a data structure and when it reaches the point, we treat each request individually, here is where GPU comes into the actions. We have a globally stored data structure which keeps information about the level and could be accessed simultaneously. So we spawned appropriate amount of thread to treat each request separately and figured out whether those request would force us to re-compute our stored information or not. 3-level BFS can be implemented to calculate new information but that is very recent discovery [4]. To decide whether certain deletion are safe or not, we will use algorithm shown in figure 1.

We will check the levels of associated vertices and determine the vertex with higher level value. After that go through the global structure keeping information about the neighbors and check if that vertex's level was determined by the connection that got terminated or not. If yes, then we will have to re-compute, in other case we are safe to remove that edge.

In real graphs, generally vertices have degree of order of 10s. So deleting some edges doesn't force us to re-computation. So this implementation rules out most of the requests.

Insertions could be treated same as Shiloach and Even's method where we keep information about component id.

In the file attached with the report we have provided code of Intuitive_approach.c, Seq.c (sequential), and Gpu_ConnectedComponent.cu for GPU implementation. We have attached results for sequential and GPU implementation only, since intuitive approach's results are not much of a use to make comparisons.

6. Results:

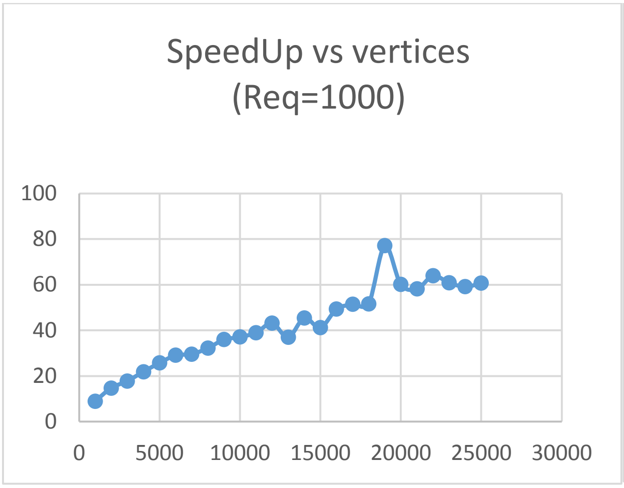
In the attached file there are 5 .csv files, in which 2 of them with old tag on them have results for various requests size set and lesser variety on number of vertices. Other 2 files have newer data with request size set= 500 and 1000 and vertex set=1000, 2000 …25000. For analysis purposes, there is file name Combined results which has comparison data and graphs.

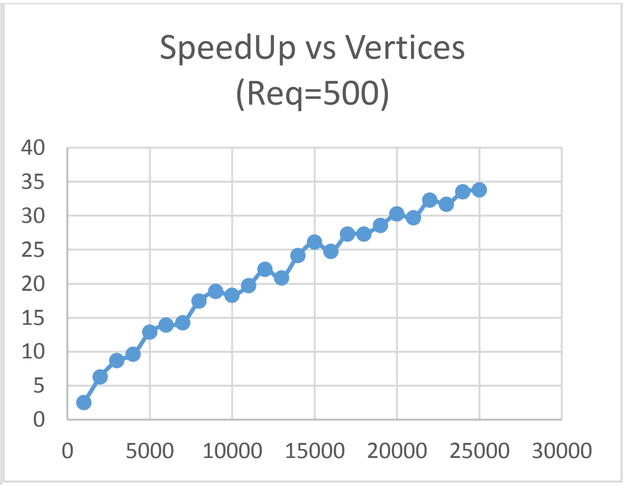
To get better idea about the speedups, we have analyzed data set with high requests and more variety in number of vertices. We have added a small table to get some idea about the results to get idea about the graphs.(To get full results, take a look at .csv file.)

Vertices	Speedup_500	Speedup_1000
1000	2.5	8.9
5000	12.9	25.12
8000	17.5	32.8
10000	18.3	37
15000	26.11	41
18000	27.30	51.6
20000	30.21	60
25000	33.78	60.68

From the results shown in the table, we can see that with the increasing size of vertex set, speed up is increasing in both the cases. Understandably speed up of 500 request set is lesser than 1000, since time taken is directly proportional to amount of request entertained.

We have derived graph on basis of these results,





During the experiments, we have taken 8- degree graph models. Since, we were making experiments on single GPU core, the vertex set's size are limited to 25000, though with collaboration and message passing, one can achieve these results for massive graphs too. With an introduction to parallel BFS, these speedup could be on bit higher side but the aim of the experiment was mainly to detect the safe edge deletion. These results are better than the results achieved by [1], though the direct comparison can't be made, since they have never explicitly shown that the speed ups achieved were for re-computation or just the detection and what technologies they have used during the experiments.

For more reference to their result, please refer [1].

7. Conclusion:

This experiment is done on low memory model without using message passing mechanism between more than one GPU processors, so results would be better in comparison what can be achieved in real life systems. In message passing, interaction between processors consumes considerable amount of time. Implementation of this algorithm will still show the improvement on results if applied properly to real systems.

My aim was to reach super-linear speed up but there is no measure determine the type of speedup, due to amount of processors used.

Scalability of this approach is still remain unknown, though we think that it still would show better results than the existing ones. We have come to realize that the most time taken is for the first iteration of BFS algorithm, so if we can implement [4] then we can test our algorithm for more realistic data sets and those results would be of more use, which would be the extension of this project if I ever choose to go forward this project.

8. References:

[1] Robert McColl, Oded Green, David A. Bader, “A New Parallel Algorithm for Connected Components in Dynamic Graphs”.
http://www.cc.gatech.edu/grads/o/ogreen3/_docs/2013DynamicConnectedComponents.pdf

[2] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig, “Sparsification-a technique for speeding up dynamic graph algorithms,” Journal of the ACM (JACM), vol. 44, no. 5, pp. 669–696, 1997.

[3] David Bader, Jonathan Berry, Adam Amon, Charles Hastings, Kamesh Madhuri, Steven Poulos, “STINGER: Spatio-Temporal Interaction Networks and Graphs (STING) Extensible Representation”.

http://cass-mt.pnnl.gov/docs/pubs/pnnlgeorgiatechsandiastinger-u.pdf

[4] Lijuan Luo, Martin Wong, Wen-mei Hwu, “An Effective GPU Implementation of Breadth-First Search”.

http://impact.crhc.illinois.edu/shared/papers/effective2010.pdf

[5] Y. Shiloach and S. Even, “An on-line edge-deletion problem,” J. ACM, vol. 28, pp. 1–4, January 1981

[6] D. Ediger, K. Jiang, J. Riedy, and D. Bader, “Massive streaming data analytics: A case study with clustering coefficients,” in Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW).