Brief Announcement: Non-blocking Dynamic Unbounded Graphs with Wait-Free Snapshot

Gaurav Bhardwaj(✉), Sathya Peri, and Pratik Shetty

Indian Institute of Technology, Hyderabad, India
{CS19RESCH11003,ai21mtech12005}@iith.ac.in, sathya_p@cse.iith.ac.in

Abstract. In this paper, we have implemented a dynamic unbounded concurrent graph which can perform the add, delete or lookup operations on vertices and edges concurrently and are linearizable. In addition to these operations, we also have a wait-free graph snapshot method. To the best of our knowledge, we are the first to develop a wait-free graph snapshot algorithm.

Keywords: Graphs · Snapshot · Lock-Free · Wait-Free

1 Introduction

Graph data structure have several real-life applications such as blockchains, maps, machine learning applications, biological networks, social networks, etc. A paired entity relation in a graph displays the relationship and structure between the objects. Social networks, for instance, use graphs to depict user relationships, which aids in making suggestions, spotting trends, and forecasting user behaviour. Over other data structures like linked lists, hash tables, trees, etc., graphs have a significant advantage in terms of application domains, making graph problem solving a major area of research.

Due to these practical applications, there has been a lot of interest on concurrent graph implementations [1,2,5]. Most of these implementations support two kinds of operations: (a) *graph-point* methods, which are adding/removing/looking-up vertices/edges on the graph. These operations can be considered as operating on one (or two) vertex points of interest. (b) *graph-set* method(s), which involves taking a partial or complete snapshot of the graph. graph-set operation consider and collect several vertices. We use the term graph-set and snapshot interchangeably.

It has been observed that constructing (partial) snapshots of a dynamic, concurrent graph efficiently is an important problem which can be used for various graph analytics operations as shown by [1]. Among the various concurrent graph structures proposed in the literature, none support *wait-free*[1] snapshot construction for unbounded graphs.

[1] A progress condition in which every thread invoking a method will complete in finite number of steps [4].

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
S. Dolev and B. Schieber (Eds.): SSS 2023, LNCS 14310, pp. 106–110, 2023.
https://doi.org/10.1007/978-3-031-44274-2_9

1.1 Our Contribution

This paper addresses this shortcoming by developing a concurrent graph structure that supports wait-free snapshot construction while the graph-point methods are lock-free. To illustrate the usefulness of the snapshot constructed, we use it to compute analytics operations Betweenness Centrality (BC) and Diameter (Dia).

Our solution is an extension of Chatterjee et al.'s [2] concurrent framework for unbounded graphs. We extend their graph-point methods for constructing a wait-free snapshot of the graph, which is based on the snapshot algorithm of Petrank and Timnat [6] developed for iterators.

2 Preliminaries and ADT

We created a concurrent lock-free graph data structure that maintains the vertices and edges in an adjacency list format inspired by Chatterjee et al's [2] implementation. The adjacency lists are maintained as lock-free linked lists.

In addition to the graph-point methods of [2], our implementation supports the following graph-set methods:

1. **Snapshot**: Given a graph G, returns a consistent state of the graph.
2. **Diameter**: Given a graph G, returns the shortest path with respect to the total number of edges traversed for two farthest nodes from all pair of vertices $u, v \in V$.
3. **Betweenness Centrality**: Given a graph G, returns a vertex which lies most frequently in the shortest path of all pair of vertices $u, v \in V$.

2.1 The Abstract Data Type (ADT)

We define an *ADT* $\mathcal{A}$ to be the collection of operations: $\mathcal{A}$ = AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge, Snap, BetweenCentrality, Diameter.

1. AddVertex (v): adds a vertex v to V ($V \leftarrow V \cup v$) if $v \notin V$ and returns `VERTEXADDED`. If $v \in V$ then returns `VERTEX ALREADY PRESENT`.
2. RemoveVertex (v): removes a vertex v from V if $v \in V$ and returns `VERTEXREMOVED`. If $v \notin V$ then returns `VERTEX NOT PRESENT`.
3. ContainsVertex (v): returns `VERTEX PRESENT` if $v \in V$ otherwise returns `VERTEX NOT PRESENT`.
4. AddEdge (u,v): returns `VERTEX NOT PRESENT` if $u \notin V \vee v \notin V$. If edge $e(u, v) \in E$, it returns `EDGE PRESENT` otherwise, it adds an edge $e(u, v)$ to E ($E \leftarrow E \cup e(u.v)$) and returns `EDGE ADDED`.
5. RemoveEdge (u,v): returns `VERTEX NOT PRESENT` if $u \notin V \vee v \notin V$. If edge $e(u, v) \notin E$, it returns `EDGE NOT PRESENT`; otherwise, it removes the edge $e(u, v)$ from E ($E \leftarrow E - e(u, v)$) and returns `EDGE REMOVED`.
6. ContainsEdge (u,v): returns `VERTEX NOT PRESENT` if $u \notin V \vee v \notin V$. If edge $e(u, v) \notin E$, it returns `EDGENOTPRESENT` otherwise, it returns `EDGE PRESENT`.

7. Snap: returns the previously described consistent snapshot of the graph.
8. BetweenCentrality: returns the Between Centrality of Graph G as described above.
9. Diameter: returns Diameter of graph G as mentioned above.

3 Design and Algorithm

We utilised the same graph structure of adjacency lists with lock-free linked lists as Chatterjee et al. [2] employed. We have separated the operations into two categories for clarity: a) graph-point operation and b) graph-set operation. Graph-point operations are comparable to those implemented by Chatterjee et al. [2], with modest adjustments to allow for more advanced wait-free graph analytics procedures. Graph-set operation necessitates a consistent snapshot of the graph, which is inspired by Timnak and Shavit's [7] iterative wait-free snapshot approach.

3.1 Graph Point Operations

We used the lock-free linked list [3] structure for defining the graph's nodes and edges. Vertices are linked lists, and each vertex is connected to the edge linked list. We modified the graph-point operation compared to the version of Chatterjee et al. [2] because we forward the value to the concurrent ongoing snapshot operation for each graph-point operation. When a point operation reads or updates a vertex or an edge, the value is forwarded to the concurrent snapshot operation for the consistent snapshot.

3.2 Graph Snapshot Operation

Timnak inspires our graph snapshot and Shavit's [7] iterator snapshot algorithm. We used the same forwarding principle, where we forward the value as reports to the snapshot operation if some concurrent snapshot operation occurs. The snapshot procedure initially gathers all the graph elements by traversing all its components. Meanwhile, all concurrent graph-point operations transfer the values of the element they act on to the snapshot method. After gathering all the data, items from the graph are added or removed based on the reports obtained during that period to generate a consistent picture.

4 Experiments and Results

Platform Configuration: We conducted our experiments on a system with Intel(R) Xeon(R) Gold 6230R CPU packing 52 cores with a clock speed of 2.10 GHz. There are two logical threads for each core, each core with a private 32 KB L1 and 1024 KB L2 cache. The 36608KB L3 cache is shared across the cores. The system has 376 GB of RAM and 1 TB of hard disk. It runs on a 64-bit Linux operating system.

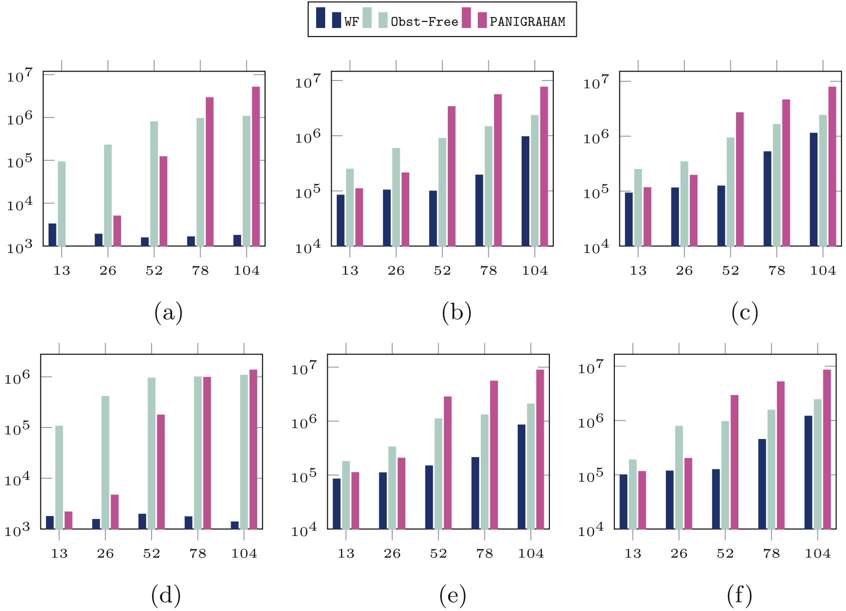

Fig. 1. Performance of our implementation compared to its counterparts. x-axis: Number of threads. y-axis: Average Time taken in microseconds.(a) Read Heavy workload with snapshot, (b) Read Heavy workload with Diameter, (c) Read Heavy workload with Betweenness Centrality, (d) Update Heavy with snapshot, (e) Update Heavy with Diameter, (f) Update Heavy with Betweenness Centrality.

Experimental Setup: All implementations are in C++ without garbage collection. We used Posix threads for multi-threaded implementation. We initially populated the graph with uniformly distributed synthetic data of 10K nodes and 20K edges for the experiments. In all our experiments, we have considered all the point operations AddVertex, RemoveVertex, ContainsVertex, AddEdge, RemoveEdge, ContainsEdge from ADT and one of the graph analytics operations from Snap, BetweenCentrality and Diameter. The evaluation metric used is the average time taken to complete each operation. We measure the average time w.r.t increasing spawned threads.

Workload Distribution : The distribution is over the following ordered set of Operations (AddVertex, RemoveVertex, ContainsEdge, AddEdge, RemoveEdge, ContainsEdge, and Critical Operation(Snap/ Diameter/ BetweenCentrality).

1. Read Heavy Workload : 3%, 2%, 45%, 3%, 2%, 45% , 2%
2. Update Heavy Workload: 12%, 13%, 25%, 13%, 12%, 25% , 2%

Algorithms : We compare our wait-free Snap/ Diameter/ BetweenCentrality approaches to the obstruction-free implementation of the same operations using Chatterjee et al. [2], and Chatterjee et al. [1]. We have named them `Obst-Free` and `PANIGRAHAM`, respectively, and our approach as `WF`.

Performance for various Graph Analytics Operation In Fig. 1, we compare the average time of the algorithms under the two different workloads mentioned above. Initially, with Snap, and then we replace the Snap operation with Diameter and BetweenCentrality. In the case of Snap, our algorithm outperforms all its counterparts by up to two orders of magnitude because if a new thread is required to execute Snap, it assists the current Snap if it is there and collaboratively finds the Snap. Thus we see that the average time remains the same even with increasing active threads as more threads will be involved in creating a snapshot. On the other hand, in the obstruction-free algorithm, each thread creates its own independent Snapshot. Each thread performs the Diameter and BetweenCentrality independently using the snapshot in all three algorithms. Hence we see the Average time increasing with threads.

References

1. Chatterjee, B., Peri, S., Sa, M., Manogna, K.: Non-blocking dynamic unbounded graphs with worst-case amortized bounds. In: International Conference on Principles of Distributed Systems (2021)
2. Chatterjee, B., Peri, S., Sa, M., Singhal, N.: A simple and practical concurrent non-blocking unbounded graph with linearizable reachability queries. In: ICDCN 2019, Bangalore, India, 04–07 January 2019, pp. 168–177 (2019)
3. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45414-4_21
4. Herlihy, M., Shavit, N.: On the nature of progress. In: OPODIS, pp. 313–328 (2011)
5. Kallimanis, N.D., Kanellou, E.: Wait-free concurrent graph objects with dynamic traversals. In: OPODIS, pp. 1–27 (2015)
6. Petrank, E., Timnat, S.: Lock-free data-structure iterators. In: Afek, Y. (ed.) DISC 2013. LNCS, vol. 8205, pp. 224–238. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41527-2_16
7. Timnat, S., Braginsky, A., Kogan, A., Petrank, E.: Wait-free linked-lists. In: OPODIS, pp. 330–344 (2012)